ABSTRACT

Title of dissertation: DATAFLOW INTEGRATION AND SIMULATION TECHNIQUES FOR DSP SYSTEM DESIGN TOOLS

Chia-Jui Hsu
Doctor of Philosophy, 2007

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and
Computer Engineering

System-level modeling, simulation, and synthesis using dataflow models of computation are widespread in electronic design automation (EDA) tools for digital signal processing (DSP) systems. Over the past few decades, various dataflow models and techniques have been developed for different DSP application domains; and many system design tools incorporate dataflow semantics for different objectives in the design process. In addition, a variety of digital signal processors and other types of embedded processors have been evolving continuously; and many off-the-shelf DSP libraries are optimized for specific processor architectures.

To explore their heterogeneous capabilities, we develop a novel framework that centers around the *dataflow interchange format* (DIF) for helping DSP system designers to integrate the diversity of dataflow models, techniques, design tools, DSP libraries, and embedded processing platforms. The dataflow interchange format is designed as a standard language for specifying DSP-oriented dataflow graphs, and the DIF framework is developed to achieve the following unique combination of

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Dataflow Integration and Simulation Techniques for DSP System Design Tools	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,Department of Electrical and Computer Engineering,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**223**	

objectives: 1) developing dataflow models and techniques to explore the complex design space for embedded DSP systems; 2) porting DSP designs across various tools, libraries, and embedded processing platforms; and 3) synthesizing software implementations from high-level dataflow-based program specifications.

System simulation using synchronous dataflow (SDF) is widely adopted in design tools for many years. However, for modern communication and signal processing systems, their SDF representations often consist of large-scale, complex topology, and heavily multirate behavior that challenge simulation — simulating such systems using conventional SDF scheduling techniques generally leads to unacceptable simulation time and memory requirements. In this thesis, we develop a *simulation-oriented scheduler* (SOS) for efficient, joint minimization of scheduling time and memory requirements in conventional single-processor environments.

Nowadays, multi-core processors that provide on-chip, thread-level parallelism are increasingly popular for the potential in high performance. However, current simulation tools gain only minimal performance improvements due to their sequential SDF execution semantics. Motivated by the trend towards multi-core processors, we develop a novel *multithreaded simulation scheduler* (MSS) to pursue simulation runtime speed-up through multithreaded execution of SDF graphs on multi-core processors. Our results from SOS and MSS demonstrate large improvements in simulating real-world wireless communication systems.

DATAFLOW INTEGRATION AND SIMULATION TECHNIQUES FOR DSP SYSTEM DESIGN TOOLS

by

Chia-Jui Hsu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Gang Qu
Professor Peter Petrov
Professor Manoj Franklin
Professor Chau-Wen Tseng

Dedication

To my Mother, Father, Brother, and Chi-Hsuan

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Shuvra S. Bhattacharyya, for his invaluable support and guidance in my M.S. and Ph.D. studies. Looking back, when I was frustrated in seeking research opportunities, he gave me a chance, and later on, provided me great opportunities to participate in two funded research projects. He guided me the way to do research, gave me enormous freedom to solve the proposed problems, and encouraged me all the time. His extraordinary editing skills also made my writings become professional publications.

I also want to thank my committee members, Professor Qu, Professor Petrov, Professor Franklin, and Professor Tseng, for their insightful feedbacks.

The research reported in this thesis was supported in part by DARPA (contract #DAAH01-03-C-R236 via the U.S. Army Aviation and Missile Command and through MCCI); the Semiconductor Research Corporation (contract #2001-HJ-905); and Agilent Technologies.

I would like to thank Chris Robbins and Carl Ecklund from Management Communications and Control, Inc. for the cooperation in the DIF project. I want to thank Jorn Janneck from Xilinx for his initiation of novel ideas. I also want to thank Jose Luis Pino, John Baprawski, Suren Ramasubbu, Junaid Khan, and Johnson Kin from Agilent Technologies for their help in the SDF simulation project. Especially, I would like to acknowledge Jose for his initiation of problems that turn out to be

two important chapters in this thesis, and thank John for the great opportunity to continue my research.

It has been a pleasure to interact with many students and graduates in the DSPCAD group, digital signal processing lab, and embedded systems research lab, including Ming-Yung, Chung-Ching, Ruirui, Ivan, Dong-Ik, Vida, Sankalita, Mainak, Neal, Hojin, William, Sebastian, Ankush, Fuat, Celine, Lin, Chaiyod, Wipawee, and Kee. I also want to thank my friends from University of Maryland, National Taiwan University, and other places for the joyful time we have been together.

Most importantly, I want to take this opportunity to thank my Mother Mei-Chu Yu and my Father Jeng-Shung Sheu for their endless love, support, and patience in my life. I gratefully dedicate this thesis and my Ph.D. degree to my parents and want to make them proud of me. I wish my brother Yao-Wen Hsu to have a great future. Finally, I really want to thank Chi-Hsuan Yin for giving me such joyful and wonderful life.

Table of Contents

List of Figures

GLOSSARY

$adj(v)$	The adjacent actors of an actor v.
$buf(e)$	The buffer size required for an SDF edge e.
$CM(c)$	The cycle mean of a cycle c in an HSDF graph.
$cns(e)$	The consumption rate of an SDF edge e.
$ct(v,t)$	The count of complete firings of an actor v up to time t since the graph starts execution.
$del(e)$	The number of initial tokens on an SDF edge e.
gcd	Greatest common divisor.
$in(v)$	The set of input edges of an actor v.
$MC(G)$	The multirate complexity of an SDF graph G.
$MCM(G)$	The maximum cycle mean of an HSDF graph G.
$out(v)$	The set of output edges of an actor v.
$prd(e)$	The production rate of an SDF edge e.
$pre(v)$	The predecessors of an actor v.
$\boldsymbol{q}_G[v]$	The repetition count of an actor v in the SDF graph G.
$snk(e)$	The sink actor of an SDF edge e.
$src(e)$	The source actor of an SDF edge e.
$SRTP$	Sum of repetition count - execution time products.
$suc(v)$	The successors of an actor v.
$t(v)$	The execution time of an actor v.
$tok(e,t)$	The number of tokens on an SDF edge e at time t.
$tok_G(S,e,k)$	The number of tokens on an edge e in the SDF graph G immediately after the actor firing associated with firing index k in the schedule S.
$\sigma(S,k)$	The actor associated with firing index k in the schedule S.
$\tau(S,v,k)$	The firing count of actor v up to firing index k in the schedule S.
$[u,v]$	A set of edges that connect from the same source vertex u to the same sink vertex v.

Chapter 1

Introduction

1.1 Overview

Communication and digital signal processing (DSP) systems play increasingly important roles in our daily life, including various wired and wireless communication devices, e.g., cellular phones, and many types of audio, image, and video processing devices, e.g., MP3 players, digital cameras, and camcorders. A significant amount of these electronic devices fall into the category of embedded systems, where combinations of hardware (e.g., microcontrollers, programmable digital signal processors (PDSPs), field programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), and other types of embedded processors) and software (e.g., embedded operating systems, drivers, codecs, DSP functions, and other types of embedded software) are designed jointly to perform dedicated functionalities.

For embedded systems in the areas of communication and DSP applications (which we refer to henceforth as *embedded DSP systems*), the growing demands for high performance, increased functionality, low power, low cost, and short time-to-market make the design space more and more complex. Due to such large complexity, the design of modern embedded DSP systems is usually aided by a variety of electronic design automation (EDA) tools [72]. In general, different EDA tools aid different phases of the design process, ranging from physical layout, logical verifica-

tion, all the way up to system-level (or high-level) modeling, simulation, and synthesis. Particularly, for system- or high-level design, model-based design methodologies are widely used in EDA tools, e.g., Simulink from Mathworks, Advanced Design System (ADS) from Agilent Technologies, LabVIEW from National Instruments, and Ptolemy II from U.C. Berkeley, to name a few. In model-based design methodologies, design representations in terms of formal models of computation (MoC) are used to capture, analyze, simulate, and in some cases, optimize and synthesize the targeted applications.

Dataflow has proven to be a useful model of computation in DSP system design [7, 37, 44, 74]. Modeling communication and signal processing systems through coarse-grain dataflow graphs is widespread in the DSP design community. Various dataflow models have been presented for different types of DSP applications, e.g., synchronous dataflow (SDF) [51], cyclo-static dataflow (CSDF) [10], multi-dimensional synchronous dataflow (MDSDF) [62], and parameterized dataflow [3]. Furthermore, many scheduling and optimization techniques have been developed in these models for different aspects of DSP design — e.g., see [7], [60], and [38].

A variety of commercial and research-oriented EDA tools incorporate dataflow semantics (mainly SDF or its closely related models), including ADS from Agilent [67], the Autocoding Toolset from MCCI [70], CoCentric System Studio from Synopsis [14], Compaan from Leiden University [76], Gedae from Gedae Inc., Grape from K. U. Leuven [48], LabVIEW from National Instruments [2], MLDesigner from MLDesign Technologies, Inc., PeaCE from Seoul National University [77], and Ptolemy II from U. C. Berkeley [21]. In general, these tools provide graphical design

environments and are developed for various primary objectives, e.g., simulation vs. synthesis; they use different specification formats, provide different sets of functional libraries, and target different sets of embedded processing platforms.

Among various embedded processing platforms, digital signal processors (e.g., those available from Texas Instruments and Analog Devices), FPGAs (e.g., those available from Xilinx and Altera), and other types of embedded processors are widely used in many embedded DSP systems. Their architectures are generally vendor-dependent or application-specific in nature, and many PDSP and FPGA vendors and third-party companies provide DSP functions and IP modules that are optimized for specific architectures and design requirements, e.g., TI DSP libraries [80, 79] and Xilinx IP cores [83].

System- or high-level modeling, simulation, and synthesis are the key features provided by model-based EDA tools for embedded system design. With the support of design environments and component libraries, designers can easily design algorithms and construct architectures for the targeted applications within short time. Formal models of computation then capture the design semantics, and system-level simulation based on the formal model is in general the most major capability provided by these tools. System simulation verifies the correctness of algorithms and architectures in the early design stage, and further analysis, optimization, and trade-offs can be performed iteratively based on the simulation results. Some design tools also provide synthesis capabilities for automatic generation of C-code implementations for PDSPs or other types of embedded processors, or for Verilog/VHDL implementations on FPGAs.

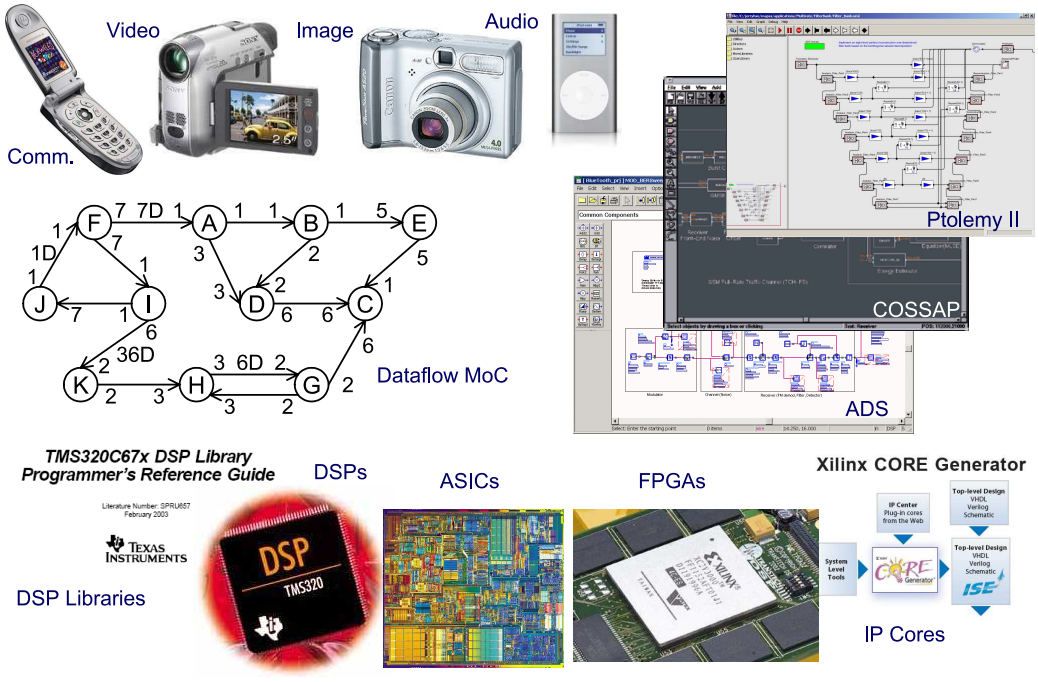


Figure 1.1: Overview of DSP system design.

Figure 1.1 presents an overview of the aforementioned scenarios. In this thesis, we focus on the *integration* perspective of DSP system design through dataflow techniques; we also address advanced *simulation* techniques in the dataflow domain for modern communication and signal processing systems.

1.2 Contributions of this Thesis

1.2.1 Dataflow Interchange Format Framework

All of the aforementioned dataflow models, EDA tools, DSP libraries, and embedded hardwares play important roles in different phases of DSP system design, and their heterogeneous capabilities introduce a large design space. Integrating these heterogeneous capabilities is beneficial because of their complementary features, e.g.,

simulation vs. synthesis, hardware vs. software support, optimization trade-offs, etc. Motivated by this perspective, we have developed the *dataflow interchange format framework* for integrating various dataflow models, techniques, design tools, DSP libraries, and embedded processing platforms.

In particular, the dataflow interchange format (DIF) [35, 31] has been designed as a standard language for specifying DSP-oriented dataflow graphs; and the *DIF package* — the software tool providing dataflow graph library, algorithm implementations, and infrastructure for porting and software synthesis — has been developed and continues to evolve for experimenting with dataflow models and techniques, and working with DSP applications across the growing family of design tools, libraries, and embedded processing platforms.

1.2.2 Porting DSP Designs

Migrating or developing DSP designs across multiple tools and libraries often becomes desirable due to their complementary capabilities, even though the heterogeneity makes it very challenging. Such migration typically requires tedious effort and is highly error-prone. Porting DSP applications across design tools and libraries is a powerful feature if it is attained through a high degree of automation, and a correspondingly low level of manual fine-tuning. When comprehensively supported, this portability is equivalent to porting across all underlying embedded processing platforms supported by these tools and libraries.

This prospect motivates a new *DIF-based porting methodology* [33] that we de-

velop through the dataflow information captured by the DIF language, and through additional infrastructure for converting dataflow-based application models to and from DIF, as well as for mapping tool-specific actors based on specifications in our novel *actor interchange format* [33]. The key idea behind the porting methodology is that except for actor information, dataflow semantics for a DSP application remains the same in DIF regardless of which design tool is used to generate it, and furthermore, porting DSP applications can be achieved by properly mapping the tool-dependent actors, while transferring the dataflow semantics unaltered.

With the DIF-based porting methodology and the porting infrastructure developed in the DIF package, migrating or developing DSP designs across tools and libraries can be achieved in a systematic manner. Through a case study of a synthetic aperture radar (SAR) application, we demonstrate the efficiency and the high degree of automation offered by our porting methodology.

1.2.3 Software Synthesis from Dataflow Models

Since the DIF language is designed as an interchange format as well as a dataflow programming language (in particular, it is designed to be read and written intuitively by designers, not just to be generated and parsed by tools), software synthesis capability provides a new path to implementation from high-level dataflow-based programming. We have developed the *DIF-to-C software synthesis framework* [36] for automatically generating monolithic C-code implementations from DSP system specifications that are programmed in DIF.

In particular, our DIF-to-C software synthesis framework integrates a significant amount of scheduling, buffering, and code generation techniques, and allows designers to associate dataflow actors with their desired C functions either designed by themselves or obtained from existing libraries. Because most programmable digital signal processors and other types of embedded processors provide C compilers, and furthermore, many PDSP vendors and third-party companies provide hand-optimized C libraries, the DIF-to-C framework offers a valuable, vendor-neutral link between formal, domain-specific DSP design and coarse grain dataflow optimizations with hand-optimized libraries and processor/platform-specific compiler optimization techniques. Furthermore, because the DIF package implements a variety of algorithms and is open for integration of new techniques, this framework allows designers to efficiently explore the complex range of implementation trade-offs that are available through various dataflow-based methods for software optimization.

Embedded hardware/software synthesis has been addressed extensively in the literature. In contrast to this prior work, the synthesis counterpart in the DIF framework emphasizes the "integration" perspective, where interoperability of semantics and methods across tools and libraries is a key objective. In this thesis, we demonstrate the novel capabilities offered by the DIF-to-C software synthesis framework through experiments that involve synthesis of several DSP applications, including CD/DAT sampling rate conversion systems, filter banks, SAR, and JPEG subsystem.

1.2.4 Efficient Simulation of Critical Synchronous Dataflow Graphs

Our work also focuses on the simulation context, which is relatively unexplored in any explicit sense in the dataflow domain. For system simulation, simulation time (including static scheduling at compile-time; and overall execution, with dynamic scheduling in some cases, during run-time) is the primary metric, while memory usage (including memory for buffering and for the schedule) must only be managed to fit the available memory resources. These considerations are quite different compared to the conventional synthesis context, where memory requirements are often of critical concern, while tolerance for compile time is relatively high [56].

System simulation using synchronous dataflow (SDF) is widespread in EDA tools for DSP system design. SDF representations of modern communication and signal processing systems typically result in *critical* SDF graphs — they consist of hundreds of components (or more) and involve complex inter-component connections with highly multirate relationships (i.e., with large variations in average rates of data transfer or component execution across different subsystems). Simulating such systems using conventional SDF scheduling techniques generally leads to unacceptable simulation time and memory requirements on modern workstations and high-end PCs.

We have developed a novel *simulation-oriented scheduler* [37, 38] , called SOS, that strategically integrates several graph decomposition and SDF scheduling techniques to provide effective, joint minimization of time and memory requirements for simulating large-scale and heavily multirate SDF graphs. We have also imple-

mented SOS in the *Advanced Design System* (ADS) from Agilent Technologies [67]. Our results from this implementation demonstrate large improvements in terms of scheduling time and memory requirements in simulating real-world, large-scale, highly-multirate wireless communication systems (e.g. 3GPP, Bluetooth, 802.16e, CDMA 2000, XM radio, EDGE, and Digital TV).

1.2.5 Multithreaded Simulation of Synchronous Dataflow Graphs

Nowadays, multi-core processors are increasingly popular desktop platforms for their potential performance improvements through on-chip, thread-level parallelism. This type of on-chip, thread-level parallelism can be further categorized into chip-level multiprocessing (CMP) [29] (e.g., dual-core or quad-core CPUs from Intel or AMD) and simultaneous multithreading (SMT) [20] (e.g., hyper-threading CPUs from Intel). However, without novel scheduling and simulation techniques that explicitly explore thread-level parallelism for executing SDF graphs, current EDA tools gain only minimal performance improvements from these new sets of processors. This is largely due to the sequential (single-thread) SDF execution semantics that underlies these tools.

Motivated by the trend towards multi-core processors, we have also developed a *multithreaded simulation scheduler*, called MSS, to pursue simulation runtime speed-up through multithreaded execution of SDF graphs on multi-core processors. MSS strategically integrates graph clustering, intra-cluster scheduling, actor vectorization, and inter-cluster buffering techniques to construct inter-thread communication

(ITC) graphs at compile-time. MSS then applies efficient synchronization and dynamic scheduling techniques at runtime for executing ITC graphs in multithreaded environments. We have also implemented MSS in the Advanced Design System. On an Intel dual-core hyper-threading (4 processing units) processor, our results from this implementation demonstrate up to 3.5 times speed-up in simulating modern wireless communication systems (e.g., WCDMA3G, CDMA 2000, WiMax, EDGE, and Digital TV).

1.3 Outline of Thesis

The organization of this thesis is as follows: We review dataflow models of computation in Chapter 2 and related work in Chapter 3. In Chapter 4, we introduce the DIF language, the DIF package, and our envisioned methodology of using DIF. Next, we present the DIF-based porting methodology in Chapter 5 and the DIF-to-C software synthesis framework in Chapter 6. In Chapter 7, we introduce the simulation-oriented scheduler, and then in Chapter 8, we present the multithreaded simulation scheduler. We conclude and discuss the future work in the final chapter.

Chapter 2

Dataflow Models of Computation

In the dataflow modeling paradigm, the computational behavior of a system is represented as a directed graph $G = (V, E)$. A vertex (node, or *actor*) $v \in V$ represents a computational module or a hierarchically nested subgraph. A directed edge $e \in E$ represents a FIFO buffer from its source actor $src(e)$ to its sink actor $snk(e)$, and imposes precedence constraints for proper scheduling of the dataflow graph. An edge e can have a non-negative integer *delay* $del(e)$ associated with it. This delay value specifies the number of initial data values (*tokens*) that are buffered on the edge before the graph starts execution. Dataflow graphs operate based on *data-driven* execution: an actor v can execute (*fire*) only when it has sufficient numbers of data values (tokens) on all of its input edges $in(v)$. When firing, v consumes certain numbers of tokens from its input edges, executes its computation, and produces certain numbers of tokens on its output edges $out(v)$.

2.1 Synchronous Dataflow

Synchronous dataflow (SDF) [51] is the most popular form of dataflow models for DSP system design. In SDF, the number of tokens produced onto (consumed from) an edge e by a firing of $src(e)$ ($snk(e)$) is restricted to be a constant positive integer that must be known at compile time; this integer is referred to as the

production rate (*consumption rate*) of e and is denoted as $prd(e)$ ($cns(e)$). We say that an edge e is a *single-rate* edge if $prd(e) = cns(e)$, and a *multirate* edge if $prd(e) \neq cns(e)$.

The constant integer restriction makes SDF very suitable for modeling multi-rate systems and benefits SDF with the compile-time capabilities such as deadlock detection, bounded memory determination, and static scheduling [7], but at the cost of limited expressive power and reconfigurability.

2.1.1 SDF Scheduling Preliminaries

Before execution, a *schedule* of a dataflow graph is computed. Here, by a schedule, we mean a sequence of actor firings or more generally, any static or dynamic sequencing mechanism for executing actors. An SDF graph $G = (V, E)$ has a valid schedule (is *consistent*) if it is free from deadlock and is sample rate consistent — that is, it has a *periodic schedule* that fires each actor at least once and produces no net change in the number of tokens on each edge [51]. In more precise terms, G is *sample rate consistent* if there is a positive integer solution to the *balance equations*:

$$\forall e \in E,\ prd(e) \times \boldsymbol{x}[src(e)] = cns(e) \times \boldsymbol{x}[snk(e)]. \tag{2.1}$$

When it exists, the minimum positive integer solution for the vector $\boldsymbol{x}$ is called the *repetitions vector* of G, and is denoted by $\boldsymbol{q}_G$. For each actor v, $\boldsymbol{q}_G[v]$ is referred to as the *repetition count* of v. A *valid minimal periodic schedule* (which is abbreviated as *schedule* hereafter in this paper) is then a sequence of actor firings in which each actor v is fired $\boldsymbol{q}_G[v]$ times, and the firing sequence obeys the data-driven properties

imposed by the SDF graph.

To provide for more memory-efficient storage of schedules, actor firing sequences can be represented through looping constructs [7]. For this purpose, a *schedule loop*, $L = (n\ T_1 T_2 \cdots T_m)$, is defined as the successive repetition n times of the invocation sequence $T_1 T_2 \cdots T_m$, where each T_i is either an actor firing or a (nested) schedule loop. A looped schedule $S = L_1 L_2 \cdots L_N$ is an SDF schedule that is expressed in terms of the schedule loop notation. If every actor appears only once in S, S is called a *single appearance schedule* (SAS), otherwise, S is called a *multiple appearance schedule* (MAS). Every valid (looped) schedule has a *unique* actor firing sequence that can be derived by unrolling all of the loops in the schedule. For example, the schedule $S = a(3b)(2a(2b)a(3b))$ represents the firing sequence *abbbabbabbbabbabbb*. Hereafter in this thesis, we assume that an SDF schedule is represented in the looped schedule format.

2.1.2 SDF Buffering Preliminaries

Although edges in an SDF graph conceptually represent FIFO buffers, implementing a FIFO structure usually leads to runtime and memory overhead due to maintaining the strict FIFO operations. In many practical implementations, only the necessary amount of memory space is allocated for dataflow edges, and edge buffers are managed between actor firings such that actor firings always access the correct subsets of live tokens.

Once a schedule is determined, buffer sizes of dataflow edges can be computed

either statically or dynamically for allocating memory space to the buffers that correspond to graph edges. In the *non-shared buffering model* [7], given a schedule S, the *buffer size* required for an edge e, $buf(e)$, is defined as the maximum number of tokens simultaneously queued on e during an execution of S, and the *total buffer requirement* of an SDF graph $G = (V, E)$ to be the sum of the buffer sizes of all edges:

$$buf(G) = \sum_{\forall e \in E} buf(e). \tag{2.2}$$

2.2 Single-Rate Dataflow and Homogeneous Synchronous Dataflow

Single-rate dataflow is a special case of SDF that models *single-rate* systems, where in single-rate systems all actors execute at the same average rate. In single-rate dataflow, the number of tokens produced onto an edge by the source actor equals to the number of tokens consumed from the same edge by the sink actor. In other words, we have $prd(e) = cns(e)$ for every edge e in single-rate graphs.

Homogeneous synchronous dataflow (HSDF) [51, 75] is a restricted form of single-rate dataflow and SDF in which every actor produces and consumes only one token from each of its input and output edges in a firing. In HSDF graphs, the production rate and consumption rate are restricted to be one on all edges. HSDF is widely used in throughput analysis and multiprocessor scheduling. Algorithms for converting between SDF, single-rate, and HSDF graphs can be found in [75]. Such conversion is illustrated in Figure 2.1.

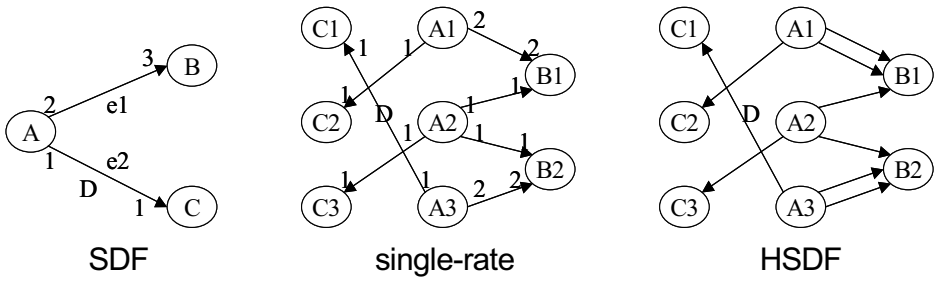


Figure 2.1: Conversion between SDF, single-rate, and HSDF.

2.3 Cyclo-Static Dataflow

In *cyclo-static dataflow* (CSDF) [10], the production rate and consumption rate are allowed to vary as long as the variation forms a fixed and periodic pattern. Explicitly, each actor v in a CSDF graph is associated with a fundamental period $\tau(v) \in Z^+$, which specifies the number of phases in one minimal period of the cyclic production/consumption pattern of v. Each time an actor is fired in a period, a different phase is executed. For each edge $e \in out(v)$, $prd(e)$ is specified as a $\tau(v)$-tuple $p_{e,1}, p_{e,2}, \ldots, p_{e,\tau(v)}$; similarly, for each $e \in in(v)$, $cns(e)$ is specified as $c_{e,1}, c_{e,2}, \ldots, c_{e,\tau(v)}$, where each $p_{e,i}$ ($c_{e,i}$) is a non-negative integer that gives the number of tokens produced onto (consumed from) e by v in the i-th phase of each period of v. CSDF offers more flexibility in representing phased behavior of an actor, but its expressive power at the level of overall individual actor functionality is the same as SDF.

2.4 Multidimensional Synchronous Dataflow

Modeling multidimensional signal processing systems by one-dimensional SDF and other stream-based dataflow models are often inefficient because streaming multidimensional data may obscure potential data parallelism and increase runtime and memory overhead in dimensional transformations. *Multidimensional synchronous dataflow* (MDSDF) [62] has been developed as an extension of SDF to better accommodate multidimensional representation.

In M-DSDF graphs, actors produce and consume M-dimensional data. For example, *2DSDF* is very suitable for modeling image processing systems where actors process images and 2-dimensional data. In M-DSDF, production rate and consumption rate are specified as M-tuples, e.g., $r_1, r_2, \ldots, r_M$, where each r_i is a positive integer that gives the size of data in the ith dimension; and dataflow semantics are now determined by the M-dimensional production rates, consumption rates, and delays.

2.5 Parameterized Dataflow

Parameterized dataflow [3] is a *meta-modeling* technique that can be applied to a variety of "base" dataflow models that have a well-defined notion of a graph iteration. Applying parameterized dataflow in this way augments the base model with powerful capabilities for dynamic reconfiguration and quasi-static scheduling. Combining parameterized dataflow with SDF forms *parameterized synchronous dataflow* (PSDF), a dynamic dataflow model that has been investigated in depth and shown

to have useful properties [3].

A PSDF actor is characterized by a set of parameters that can control the actor's functionality as well as the actor's dataflow behavior, e.g., production rate and consumption rate. A DSP application is modeled in PSDF through a *PSDF subsystem*. A PSDF subsystem consists of three PSDF graphs: the *init* graph, the *subinit* graph, and the *body* graph. Intuitively, the body graph models the main functional behavior of the specification, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the parameters. For complete syntax and semantics of PSDF modeling, we refer the reader to [3].

Chapter 3

Related Work

3.1 Dataflow Related Tools and Languages

A variety of commercial and research-oriented design tools incorporate dataflow semantics, including ADS from Agilent [67], the Autocoding Toolset from MCCI [70], CoCentric System Studio from Synopsis [14], Compaan from Leiden University [76], Gedae from Gedae Inc., Grape from K. U. Leuven [48], LabVIEW from National Instruments [2], MLDesigner from MLDesign Technologies, Inc., PeaCE from Seoul National University [77], and Ptolemy II from U. C. Berkeley [21].

Silage [27] and StreamIt [81] are two existing textual languages for designing DSP systems. DIF is different from Silage and StreamIt in its emphasis on supporting and unifying a broad range of different dataflow modeling semantics, and its emphasis on high-level dataflow-based analysis and optimization, such as analysis of interactions among dataflow production and consumption rates, and optimizations for scheduling, memory requirements, and performance.

SystemC is a C++-based modeling language/library for system level design [28]. The simulation kernel in SystemC is based primarily on discrete event semantics. Patel and Shukla [65] have extended SystemC with different simulation kernels — including kernels for communicating sequential processes, synchronous dataflow, and finite state machines — to improve simulation efficiency. Haubelt

et al. [30] recently presented a SystemC-based solution supporting automatic design space exploration, performance evaluation, and system generation for mixed hardware/software solutions mapped onto FPGA-based platforms.

Several tools provide code generation capabilities from dataflow and related models — e.g., Simulink with Real-Time Workshop from the MathWorks, and Ptolemy II from U.C. Berkeley. Zhou et al. [84] recently presented a code generation framework for actor-oriented models in Ptolemy II. This framework applies model analysis to discover data types, buffer sizes, parameter values, model structure and model execution schedules, and then applies partial evaluation on the known information to generate implementations in the target language (currently, C). However, in order to generate actor code, this framework requires the corresponding code blocks to be implemented in the target language in the same structure of the Ptolemy Java actor. In contrast, our DIF-to-C software synthesis framework [36] allows users to directly integrate arbitrary kinds of C library functions into dataflow-oriented software synthesis (see Section 6.3).

3.2 Scheduling Related Work

Various scheduling algorithms and techniques have been developed for different applications of SDF graphs. For example, Bhattacharyya et al. [7] has presented a heuristic for minimum buffer scheduling. A simpler variant of this algorithm has been used in both the Gabriel [50] and Ptolemy [13] environments, and a similar algorithm is also given in [18]. We refer to these demand-driven, minimum-buffer

scheduling heuristics as *classical SDF scheduling*. This form of scheduling is effective at reducing total buffer requirements, but its time complexity, and the lengths of its resulting schedules generally grow exponentially in the size of multirate SDF graphs.

In general, the problem of computing a buffer-optimal SDF schedule is NP-complete, and the lengths of buffer-optimal schedules usually increase exponentially in the size of the SDF graph. A single appearance schedule (SAS) [7] is often preferable due to its compact code size. A valid SAS for any consistent, acyclic SDF graph can be easily derived from the *flat strategy* [7], but at the cost of relatively large buffer requirements and latencies.

Bhattacharyya, Ko, and Murthy have developed several scheduling algorithms for joint code and data minimization in software synthesis. The *acyclic pairwise grouping of adjacent nodes* (APGAN) [7] technique is a heuristic to generate a buffer-efficient topological sort (and looped schedule) for acyclic graphs. The *dynamic programming post optimization* (DPPO) [7] performs dynamic programming over a given actor ordering (topological sort) to generate a buffer-efficient looped schedule. It can be combined with different cost functions to be adapted to different objectives — for example, GDPPO [7], CDPPO [85], and SDPPO [59]. For graphs containing cycles, the *loose interdependence algorithm framework* (LIAF) [7] has been developed for generating single appearance schedules whenever they exist. Beyond single appearance schedules, [45] presents a *recursive procedure call* (RPC) based technique such that the resulting procedural implementation is bounded polynomially in the graph size with low memory requirements. These algorithms are implemented in the DIF package [36] for synthesis of embedded DSP software. Some

of these algorithms are also integrated in the simulation-oriented scheduler in novel ways that more efficiently address the constraint of simulation efficiency.

Task-level vectorization, or block processing, is a useful dataflow graph transformation that can significantly improve execution performance by allowing subsequences of data items to be processed through individual task invocations. Block processing has been studied in single-processor software synthesis in various previous efforts, e.g., [69, 47, 46]. In contrast to these efforts, we focus in this thesis on actor vectorization techniques that are suited to multithreaded implementation contexts.

Based on developments in [11], the *cluster-loop scheduler* has been developed in the Ptolemy design tool [13] as a fast heuristic — i.e., with scheduling run-time as a primary criterion. This approach recursively encapsulates adjacent groups of actors into loops to enable possible execution rate matches and then clusters the adjacent groups. Multirate transitions, however, can prevent this method from completely clustering the whole graph. Since any un-clustered parts of the graph are left to classical SDF scheduling, this can result in large run-times and storage requirements for constructing the schedule. Our experiments in Section 7.5 demonstrate problems encountered with this method on critical graphs.

Pino, Bhattacharyya, and Lee [66] have studied hierarchical scheduling in the multiprocessor scheduling context for reducing the complexity of scheduling SDF graphs onto multiprocessors. In the DIF framework, we develop hierarchical scheduling for a different purpose in software synthesis — that is, preserving the original hierarchical structure (i.e., the design hierarchy) in the generated code.

Oh, Dutt, and Ha [64] have developed the *dynamic loop count single appearance scheduling* technique. This approach generates a looped single appearance schedule, and iteration counts of loops can be determined at run-time by evaluating statements that encapsulate states of edges. This algorithm is geared towards minimizing buffer requirements for software synthesis, but its complexity and run-time overhead are relatively high.

3.3 Buffering Related Work

The total buffer requirement defined in Equation (2.2) is based on the *non-shared memory model*, i.e., each buffer is allocated individually in memory and is live throughout a schedule. Several scheduling algorithms described above are developed for improving memory requirements based on this model. Ade et al. [1] have developed methods to compute lower bounds on buffer requirements based on analysis of each directed or undirected cycle; Geilen et al. [26] have also developed an approach to compute minimum buffer requirements based on model checking. However, the complexities of these approaches are not polynomially bounded in the graph size. As a result, they are not acceptable for the purposes that we are addressing in this thesis.

In practice, memory space can be reduced by sharing memory across multiple buffers as long as their lifetimes (at the granularity of actor firings) do not overlap, and a systematic *buffer sharing* technique has been developed in [59] based on this motivation. Furthermore, merging opportunities that are present at the

input/output buffers of individual actors are exploited by the *buffer merging* technique [6], which is based on a form of actor characterization called the *CBP* (consumed before produced) parameter. The CBP parameter characterizes the lifetimes of individual tokens at the granularity of an actor invocation. Murthy and Bhattacharyya [60] then present an integrated approach that combines both techniques (sharing and merging) to explore buffer minimization opportunities at both levels. These techniques have been shown to produce significant memory reductions over the non-shared memory model. However, due to a general lack of pre-defined CBP characterizations for off-the-shelf DSP libraries, we only implement the buffer sharing technique in the DIF-to-C framework. In addition, buffer sharing and buffer merging are geared more for synthesis of streamlined embedded software, and their relatively high complexity makes them not ideally suited for our primary concerns of simulation time reduction and algorithm scalability in critical SDF graphs.

Oh, Dutt, and Ha [63] have also presented the *shift buffering* technique for buffer management. This technique shifts samples from higher buffer indices to lower indices in order to avoid wrap-around accesses in traditional circular (modulo) buffering. Wrap-around accesses prohibit using library functions that generally assume linear data storage. For efficient shifting, a given looped schedule is flattened and shift operations are inserted. However, the resulting schedule may not be able to loop back, and this can lead to an increase in code size. This shifting approach can also introduce significant run-time overhead — for example, when there are large amounts of initial delays. However, because of the novel trade-offs that it introduces, this shifting approach is generally useful to consider during software

synthesis.

3.4 Multiprocessor Related Work

Multiprocessor scheduling for HSDF and related models has been extensively studied in the literature, e.g., see [42, 73, 75, 41]. Sarkar [73] presented partitioning and scheduling heuristics that essentially apply bottom-up clustering of tasks to trade-off communication overhead and parallelism. Sriram and Bhattacharyya [75] reviewed an abundant set of scheduling and synchronization techniques for embedded multiprocessors, including various techniques for inter-processor communication conscious scheduling, the ordered-transactions strategy, and synchronization optimization in self-timed systems [8, 9].

In general, the above multiprocessor scheduling techniques work on HSDF graphs. However, converting an SDF graph to an equivalent HSDF graph can result in an exponential increase in the number of actors. Pino et al. [66] proposed a hierarchical scheduling framework that reduces the complexity of scheduling SDF graphs onto multiprocessors. The core of this framework is a clustering algorithm that decreases the number of nodes before SDF-to-HSDF transformation.

Regarding SDF scheduling specific to multithreaded simulation, the only previous work that we are aware of is the thread cluster scheduler developed by Kin and Pino [43] in Agilent ADS. This approach applies recursive two-way partitioning on single-processor schedules that are derived from the cluster loop scheduler and then executes the recursive two-way clusters with multiple threads in a pipelined fashion.

Experimental results in [43] show an average of 2 times speedup on a four-processor machine. However, according to our recent experiments, in which we used the same scheduler to simulate several wireless designs, this approach does not scale well to simulating highly multirate SDF graphs.

Chapter 4

Dataflow Interchange Format

The dataflow interchange format (DIF) [35, 31] is proposed as a standard language for specifying and integrating arbitrary dataflow-oriented semantics for DSP system design. The DIF language syntax for dataflow semantic specification is designed based on dataflow theory and is independent of any design tool. Therefore, DIF is suitable as an interchange format for different design tools that incorporate dataflow semantics because it can fully capture essential modeling information. For a DSP application, the dataflow semantic specification is unique in DIF regardless of the design tool used to originally enter the specification. Moreover, because most design tools are fundamentally based on *actor-oriented design* [53], DIF also provides syntax to specify tool-specific actor information. Although this information may be irrelevant to many dataflow-based analyses, it is essential in porting (see Chapter 5) and software synthesis (see Chapter 6).

DIF is not aimed to directly describe detailed executable code. Such code should be placed in detailed implementations (e.g., using a commercial dataflow-based design tool), or in libraries that can be optionally associated with DIF specifications (e.g., in C code libraries for DIF-to-C synthesis). Unlike other description languages or interchange formats, such as XML [82], the DIF language is also designed to be read and written by designers who wish to specify or understand

applications based on a common, unified, DSP-oriented dataflow graph notation. As a result, the language is clear, intuitive, and easy to learn and use for those who have familiarity with dataflow semantics.

In this chapter, we introduce the DIF language in Section 4.2 and illustrate how to use DIF to specify dataflow graphs in Section 4.3. We then introduce the associated DIF package in Section 4.4, and in Section 4.5, we discuss the methodology of using DIF in DSP system design.

4.1 The DIF Hierarchy

Dataflow models of computation has been reviewed in Chapter 2. For a sophisticated DSP application, the overall system is usually modeled as a hierarchical graph in which the computations associated with certain actors, called *hierarchical actors*, can be specified as nested dataflow graphs. This is a well-known approach, but the formal dataflow graph definition does not describe such hierarchical nesting. Therefore, a *hierarchy* structure is introduced in DIF for specifying hierarchical dataflow graphs. In DIF semantics, an actor can represent either an indivisible computation or a hierarchically nested subgraph (called a *supernode* in DIF).

A hierarchy $H = (G, I, M)$ consists of a graph G with an *interface* I and a set of mappings M. Suppose that a supernode s in G represents a nested sub-hierarchy $H' = (G', I', M')$, then a *refinement* $H' = M(s)$ is established for refining s to H'. The sub-hierarchy H' of s is denoted as $sub(s)$, and G' is called a *subgraph* of s. The set of mappings M can be described as a function whose domain is simply the set of

supernodes in G and whose range is obtained through the property $M(s) = sub(s)$ for every supernode s. A directed *port* p of the hierarchy H is a dataflow gateway through which tokens flow into (input port) or flow out of (output port) the graph G. The interface I is a set consisting of all ports in H. Viewed from within G, a port $p \in I$ associates with a node v in G, and this is denoted as $v = asc(p)$. Suppose that H is a sub-hierarchy represented by a supernode s'' in a higher level graph G'', i.e. $H = M''(s'')$, then viewed from outside of G, a port $p \in I$ can either connect to an edge e'' in the higher level graph G'' or connect to a port p'' in the higher level hierarchy $H'' = (G'', I'', M'')$; these are denoted as $e'' = cnt(p)$ or $p'' = cnt(p)$, respectively.

In nested hierarchical dataflow graphs, for a node associated with an output (input) port p, the production (consumption) rate of that connection is specified with p and denoted as $prd(p)$ ($cns(p)$) — this is because the edge in that connection is outside the graph. Furthermore, because production and consumption rates of a supernode depend on the repetitions vector of the subgraph, they are left unspecified and are computed during scheduling.

4.2 The DIF Language

Dataflow interchange format is designed as a standard language for specifying DSP-oriented dataflow graphs. DIF provides a unique set of semantic features to specify graph topologies, hierarchies, dataflow-related and actor-specific information. DSP applications specified by the DIF language are usually referred to as *DIF*

specifications. In particular, the DIF language grammar and the associated parser framework are developed using a Java-based compiler-compiler called *SableCC* [22]. We introduce the DIF version 0.2 language syntax, as presented in Figure 4.1, in this section. For complete DIF language grammar (in SableCC) and detailed syntax description, we refer the reader to [31].

In Figure 4.1, items in boldface are built-in keywords; non-bold items are specified by users or generated by tools; items enclosed by squares are optional; and "..." represents optionally repeated statements. The dataflow model keyword *dataflowModel* specifies the dataflow model that is used to model the application, e.g., *dif*, *sdf*, *csdf*, *mdsdf*, etc. The *graphID* specifies the name (identifier) of the dataflow graph.

The *basedon* block provides a convenient way to reuse the structure of a predefined graph *graphID*. As long as the referenced graph has compatible topology, interface, and refinement blocks, designers can simply refer to it and override the graph name, parameters, and attributes to instantiate a new graph. In many DSP applications, duplicated subgraphs usually have the same topologies but different parameters or attributes. The basedon block is designed to support this characteristic and promote conciseness and code reuse.

The *topology* block sketches the topology of a dataflow graph $G = (V, E)$. The *nodes* statement specifies *nodeID* for each node $v \in V$. The *edges* statement specifies *edgeID (sourceNodeID, sinkNodeID)* for each edge $e = (src(e), snk(e)) \in E$.

The *interface* block defines the interface I of a hierarchy $H = (G, I, M)$. The *inputs* statement defines *portID : nodeID* for each input port $p \in I$ and the inside

```
dataflowModel graphID {
    basedon {graphID;}
    topology {
        nodes = nodeID, ..., nodeID;
        edges = edgeID (sourceNodeID, sinkNodeID), ..., edgeID (sourceNodeID, sinkNodeID);
    }
    interface {
        inputs = portID [: nodeID], ..., portID [: nodeID];
        outputs = portID [: nodeID], ..., portID [: nodeID];
    }
    parameter {
        parameterID [: parameterType]; ...; parameterID [: parameterType];
        parameterID [: parameterType] = value; ...; parameterID [: parameterType] = value;
        parameterID [: parameterType] : range; ...; parameterID [: parameterType] : range;
    }
    refinement {
        subgraphID = supernodeID;
        subPortID : edgeID; ...; subPortID : edgeID;
        subPortID : portID; ...; subPortID : portID;
        subParameterID = parameterID; ...; subParameterID = parameterID;
    }
    ...
    builtInAttribute {
        [elementID] = value; ...; [elementID] = value;
        [elementID] = ID; ...; [elementID] = ID;
        [elementID] = ID, ..., ID; ...; [elementID] = ID, ..., ID;
    }
    ...
    attribute userDefinedAttribute {
        [elementID] = value; ...; [elementID] = value;
        [elementID] = ID; ...; [elementID] = ID;
        [elementID] = ID, ..., ID; ...; [elementID] = ID, ..., ID;
    }
    ...
    actor nodeID {
        computation = "stringDescription";
        attributeID [: attributeType] = value; ...; attributeID [: attributeType] = value;
        attributeID [: attributeType] = ID; ...; attributeID [: attributeType] = ID;
        attributeID [: attributeType] = ID, ..., ID; ...; attributeID [: attributeType] = ID, ..., ID;
    }
    ...
}
```

Figure 4.1: The dataflow interchange format version 0.2 language syntax.

node association $v = asc(p) \in V$. Similarly, the *outputs* statement defines each output port and its associated node. DIF permits defining an interface port without an associated node, so *nodeID* is optional.

In many DSP applications, designers often parameterize important attributes such as the order of an FFT actor. In *interval-rate, locally-static dataflow* [78], unknown production and consumption rates are specified by their minimum and maximum values. In parameterized dataflow [3], production and consumption rates are even allowed to be parameterized and dynamically determined. The *parameter* block is designed to support parameterizing values, ranges of values, and value-unspecified attributes. In a parameter definition statement, a parameter identifier *parameterID* is defined, and addition information can be given optionally in *parameterType*, e.g., the data type of a parameter. The value of a parameter is assigned in *value*, but it is not necessary because DIF permits to define a parameter alone. DIF supports various value types; these value types will be introduced shortly. DIF also supports specifying the range *range* of possible values for a parameter. A range is specified in *interval* format such as $(1, 2)$, $(3.4, 5.6]$, $[7, 8.9)$, $[-3.1\mathrm{E}^{+3}, +0.2\mathrm{e}^{-2}]$, or a set of discrete numbers such as $\{-2, 0.1, -3.6\mathrm{E}^{-9}\}$, or a combination of intervals and discrete sets such as $(1, 2) + (3.4, 5.6] + \{-2, 0.1, -3.6\mathrm{E}^{-9}\}$.

The *refinement* block is used to refine hierarchical structures. For each supernode s in a graph $G = (V, E)$, there should be a corresponding refinement block in the DIF specification to specify the supernode-subgraph refinement $H' = sub(s)$ by *subgraphID* = *supernodeID*. In addition, for every port $p' \in I'$ in sub-hierarchy $H' = (G', I', M')$, the outside connection $e = cnt(p')$ or $p = cnt(p')$ is also speci-

fied by *subPortID : edgeID* or *subPortID : portID*, where $e \in E$, $p \in I$, and $H = (G, I, M)$. Moreover, unspecified parameters (parameters whose values are unspecified) in subgraph G' can also be specified by parameters in G through *subParameterID = parameterID*.

The *built-in attribute* block is used to specify dataflow modeling information. Every dataflow model in DIF can define its own built-in attributes and its own method to process those built-in attributes. The DIF language parser treats built-in attributes in a special way such that the method defined in the corresponding parser is invoked to handle them. The keyword *builtInAttribute* points out which built-in attribute associated with the dataflow model is specified. The element identifier, *elementID*, can be a node identifier, an edge identifier, or a port identifier to which the builtin attribute belongs. *elementID* can also be left blank; in this case, the built-in attribute belongs to the graph itself. DIF supports assigning an attribute by *value* in a variety of value types, an identifier *ID*, or a list of identifiers *ID, ..., ID*.

In general, *production*, *consumption*, and *delay* are commonly-used built-in attributes for an edge in many dataflow models to specify the production rate, consumption rate, and delay associated with the dataflow edge. In hierarchical dataflow models as discussed in Section 4.1, built-in attributes *production* and *consumption* are also used for a port to specify data rates of the associated node, because such node have no edges on the corresponding connections.

The *user-defined attributes* block allows designers to define and specify their own attributes. The syntax is the same as the built-in attributes block. The only

difference is that this block starts with the keyword *attribute* followed by the user-defined attribute identifier, *userDefinedAttribute*.

The *actor* block is designed to specify tool-specific actor information. The associated *computation* is a built-in actor attribute for specifying the actor's computation in *"stringDescription"* (e.g., what the actor does, or what the associated function is). Other actor information is specified as attributes — e.g., the identifier of an actor's component such as a port, an argument, or a parameter is used as *attributeID*. Moreover, additional information of the component can be optionally given in *attributeType*, e.g., to indicate that the component is input, output, or a parameter of an actor. An actor attribute can be assigned a value *value*, or an identifier *ID* for specifying its associated element (e.g., edge, port, or parameter), or a list of identifiers *ID, ..., ID* for indicating multiple associated elements of the attribute.

DIF supports most commonly used value types in DSP operations: integer, double, complex, integer matrix, double matrix, complex matrix, string, boolean, and array. Scientific notation is supported in DIF in the double format, e.g., $+1.2\text{E}^{-3}$, -4.56e^{+7}. A complex value is enclosed by parentheses as (real part, imaginary part), and the real and imaginary parts are double values. For example, a complex value $1.2\text{E}^{-3} - 4.56\text{e}^{+7}\text{i}$ is represented as $(1.2\text{E}^{-3}, -4.56\text{e}^{+7})$ in DIF. Matrices are enclosed by brackets; "," is used to separate elements in a row; and ";" is used to separate rows, e.g., [1, 2; 3, 4]. A string value should be double quoted as "string". A boolean value is either *true* or *false*. These value types in DIF should be sufficient in most DSP applications. If a certain value type is not supported, it

can be handled to some extent by representation through the string type.

4.3 DIF Specifications for Dataflow Graphs

Note that any dataflow semantics can be specified using the *DIF model* of dataflow supported by DIF and the corresponding *DIFGraph* intermediate representation (see Section 4.4). In this DIF model, which provides the most general form of dataflow supported by DIF, the *dataflowModel* keyword is *dif*, and there is no restriction in using any syntax or semantics provided by the DIF language to describe a DIF graph. However, for performing sophisticated analyses and optimizations for a particular dataflow model of computation, it is usually useful to have more detailed and customized features in DIF that support the model. This is why support and exploration of different dataflow models for incorporation into DIF is an important area for development of the language and software infrastructure (also see Section 4.4).

The current version of the DIF language is capable of specifying synchronous dataflow (SDF) [51], single-rate dataflow, homogeneous synchronous dataflow (HSDF) [51, 75], cyclo-static dataflow (CSDF) [10], multidimensional synchronous dataflow (MDSDF) [62], parameterized synchronous dataflow (PSDF) [3], Boolean-controlled dataflow (BDF) [11], integer-controlled dataflow (IDF) [12], binary cyclo-static dataflow (BCSDF) [31], and interval-rate locally-static dataflow (ILDF) [78]. Here, we present DIF specification examples for SDF, CSDF and MDSDF. Examples for other dataflow models can be found in [31].

In SDF, the *dataflowModel* keyword is *sdf*. The three edge attributes, $prd(e)$, $cns(e)$, and $delay(e)$, are specified as SDF built-in attributes, *production*, *consumption*, and *delay*, and their values are restricted to integers. In hierarchical SDF graphs, for a node v associated with an output/input port p, the production/consumption rate of that connection is denoted as $prd(p)/cns(p)$. Since production and consumption rates of a supernode depend on the repetitions vector of the subgraph [7], they are left unspecified and are computed during scheduling. Figure 4.2 presents a tree-structured filter bank modeled in SDF. The corresponding DIF specification is shown in Figure 4.3.

In CSDF, the *dataflowModel* keyword is *csdf*, and built-in attributes *production*, *consumption*, and *delay* are specified as integer vectors. Figure 4.4 presents an up/down sampling example modeled in CSDF. The corresponding DIF specification is presented in Figure 4.5.

In MDSDF, the *dataflowModel* keyword is *mdsdf*, and built-in attributes *production*, *consumption*, and *delay* are specified as M-tuple integer vectors. Figure 4.6 presents a 2-D discrete wavelet transform modeled in MDSDF. The corresponding DIF specification is presented in Figure 4.7.

4.4 The DIF Package

The DIF package is a Java software package developed along with the DIF language. In general, it consists of three basic building blocks: the DIF front-end, the DIF representation, and algorithm implementations.

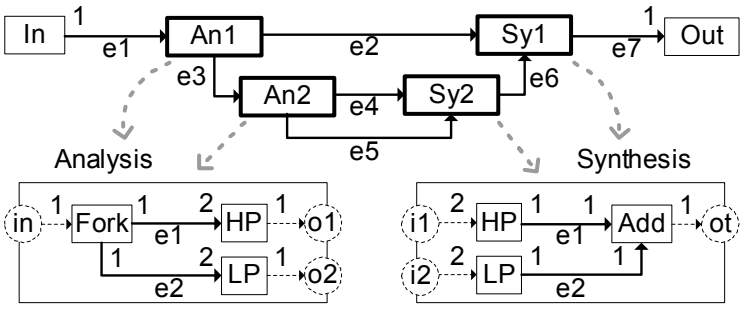


Figure 4.2: Hierarchical SDF graphs of a tree-structured filter bank. (Supernodes are shown in bold blocks; and production and consumption rates are indicated at the ends of edges and alongside ports.)

```
sdf Analysis1 {
    topology { nodes = Fork, HP, LP; edges = e1 (Fork, HP), e2 (Fork, LP); }
    interface { inputs = in : Fork; outputs = o1 : HP, o2 : LP; }
    production { e1 = 1; e2 = 1; o1 = 1; o2 = 1; }
    consumption { e1 = 2; e2 = 2; in = 1; }
    attribute datatype { e1 = "float"; e2 = "float"; in = "float"; ...; }
    actor HP { computation = "FIR"; decimation = 2; interpolation = 1; coefs = [...]; }
    ...
}
sdf Analysis2 { basedon { Analysis1; } }
sdf Synthesis1 {...}
sdf Synthesis2 { basedon { Synthesis1; } }
sdf filterBank {
    topology { nodes = In, An1, An2, Sy1, Sy2, Out; edges = e1 (In, An1), ..., e7 (Sy1, Out); }
    refinement { Analysis1 = An1; in : e1; o1 : e2; o2 : e3; }
    refinement { Analysis2 = An2; in : e3; o1 : e4; o2 : e5; }
    refinement { Synthesis1 = Sy1; i1 : e2; o2 : e6; ot : e7; }
    refinement { Synthesis2 = Sy2; i1 : e4; o2 : e5; ot : e6; }
    ...
    production { e1 = 1; }
    consumption { e7 = 1; }
    attribute datatype { e1 = "float"; ...; e7 = "float"; }
    ...
}
```

Figure 4.3: The DIF specification of Figure 4.2.

Figure 4.4: A CSDF graph of an up/down sampling example.

```
csdf upDownSampling {
    topology {
        nodes = IN, UP3, FIR, DOWN2, OUT;
        edges = e1(IN, UP3), e2(UP3, FIR), e3(FIR, DOWN2), e4(DOWN2, OUT);
    }
    production {
        e1=1; e2=[1,1,1]; e3=1; e4=[1,0];
    }
    consumption {
        e1=[1,0,0]; e2=1; e3=[1,1]; e4=1;
    }
}
```

Figure 4.5: The DIF specification of Figure 4.4.

4.4.1 DIF Representation

For each supported dataflow model, the DIF package provides an extensible set of data structures (object-oriented Java classes) for representing and manipulating dataflow graphs in the model. In the context of the DIF package, these graph-theoretic object representations for the dataflow model are referred to as the *DIF representation* of the model. The collection of all dataflow graph classes along with their associated support classes in the DIF package forms the *DIF dataflow graph library*. Figure 4.8 presents the central class hierarchy in the DIF dataflow graph library.

The *DIFGraph* is the most general graph class; It represents the basic dataflow graph structure and provides methods that are common to all models. For a more specialized dataflow model, development can proceed naturally by extending the general DIFGraph class (or suitable subclass) and overriding and adding new meth-

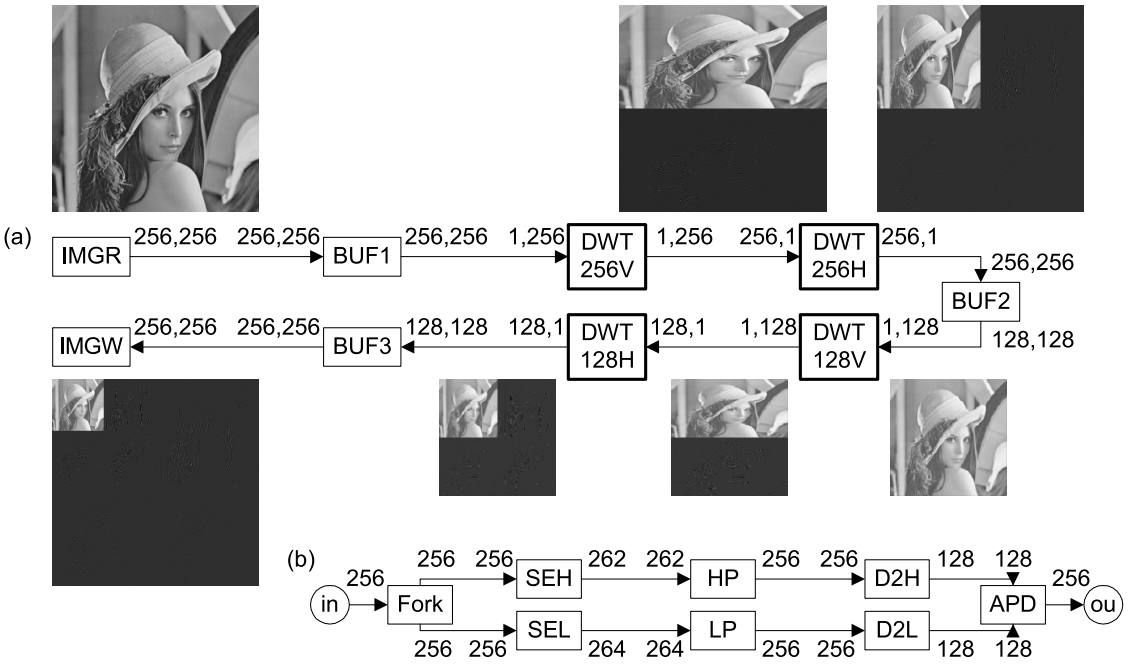


Figure 4.6: A MDSDF graph of two dimensional discrete wavelet transform. (a): 2DSDF graph of 2DDWT, where *DWT256V*, *DWT256H*, *DWT128V*, and *DWT128H* are supernodes for processing 1-D DWT for 256-pixel/128-pixel columns/rows. (b): 1DSDF subgraph of *DWT256V* and *DWT256H*.

```
mdsdf dwt256h {
    topology { nodes = HP, LP, D2H, D2L, ...; edges = e1(FRK, SEH), ..., e8(D2L, APD); }
    interface { inputs = in : FRK; outputs = out : APD; }
    production { e1 = [256]; e2 = [262]; e3 = [256]; ..., e7 = [256]; e8 = [128]; out = [256]; }
    consumption { e1 = [256]; e2 = [262]; e3 = [256]; ..., e7 = [256]; e8 = [128]; in = [256]; }
    actor HP { computation = "vsip_convolve1d_d"; ...; }
    actor LP { computation = "vsip_convolve1d_d"; ...; }
    ... }
mdsdf dwt256v { basedon { dwt256h; } }
mdsdf dwt128h { ... production { e1 = [128]; ...; } consumption { e1 = [128]; ...; } ... }
mdsdf dwt128v { basedon { dwt128h; } }
mdsdf TDDWT {
    topology { nodes = DWT256V, ...; edges = e1(IMGR,BUF1), ..., e8(BUF3,IMGW); }
    refinement { dwt256v = DWT256V; in : e2; out : e3; }
    refinement { dwt256h = DWT256H; in : e3; out : e4; }
    refinement { dwt128v = DWT128V; in : e5; out : e6; }
    refinement { dwt128h = DWT128H; in : e6; out : e7; }
    production { e1 = [256,256]; e2 = [256,256]; ..., e5 = [128,128]; e8 = [256,256]; }
    consumption { e1 = [256,256]; e4 = [256,256]; ..., e7 = [128,128]; e8 = [256,256]; }
    ... }
```

Figure 4.7: The DIF specification of Figure 4.6.

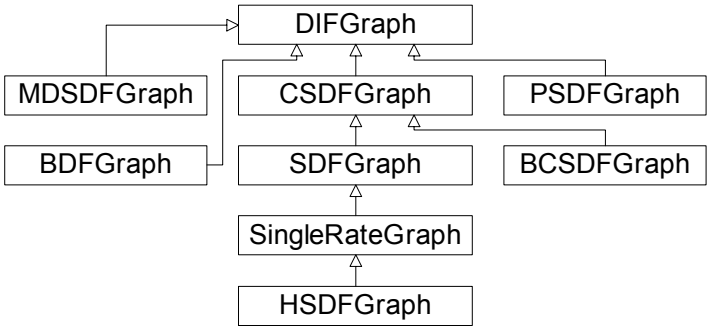


Figure 4.8: The DIF graph class hierarchy.

ods to perform more specialized functions. For example, CSDF, SDF, single-rate dataflow, and HSDF are related in a way that each succeeding model among these four is a special case of the preceding model. Accordingly, *CSDFGraph*, *SDFGraph*, *SingleRateGraph*, and *HSDFGraph* form a class hierarchy in the DIF package such that each succeeding graph class inherits from the more general one that precedes it (see Figure 4.8).

In addition to the aforementioned fundamental dataflow graph classes, the DIF package also provides *MDSDFGraph* for multidimensional synchronous dataflow, *BDFGraph* for Turing-complete *Boolean-controlled dataflow* [11], *PSDFGraph* for reconfigurable PSDF model, and *BCSDFGraph* for *binary cyclo-static dataflow* [31]. Furthermore, a variety of other dataflow models are being explored in DIF.

4.4.2 DIF Front-End

The *DIF front-end* provides an integrated interface for automatic conversion between DIF specifications (.dif files) and the DIF representations (Java dataflow graph objects). The DIF front-end consists of a *Reader* class, a set of language

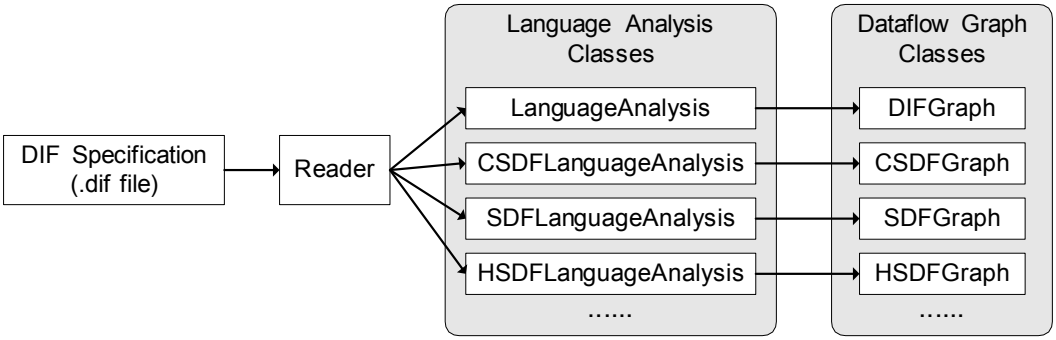


Figure 4.9: The DIF front-end reader.

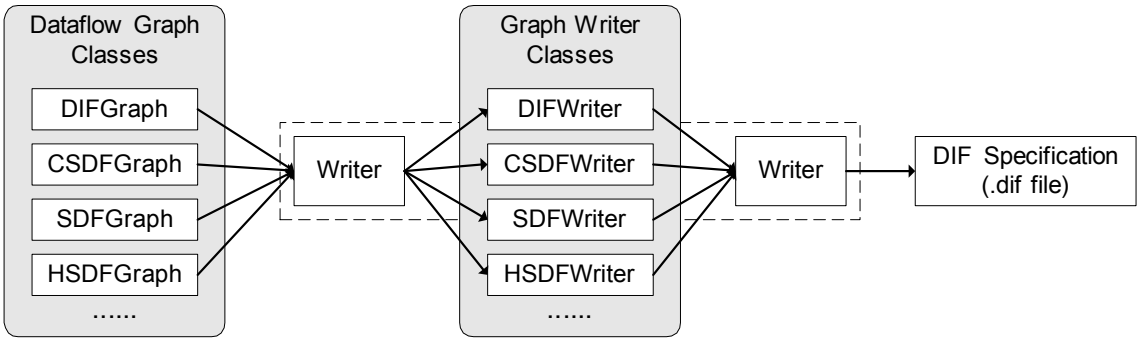


Figure 4.10: The DIF front-end writer.

parsers (*LanguageAnalysis* classes), a *Writer* class, and a set of dataflow graph writer classes. In particular, the language parser framework are automatically generated using SableCC [22], a Java-based compiler compiler.

The Reader class is the unique front-end interface that automatically constructs the corresponding DIF representation from a given DIF specification. As illustrated in Figure 4.9, *Reader* invokes the right language analysis class based on the model keyword specified in the DIF specification. On the other hand, the Writer class is the unique front-end interface to generate a DIF specification from a given DIF representation. As illustrated in Figure 4.10, *Writer* invokes the right graph writer class based on the type of the given dataflow graph object.

In our implementation, *LanguageAnalysis* is the base class to parse *DIF graph* specifications and to construct DIFGraph objects. The differences between language analysis classes are in processing model specific built-in attributes and initiating graph objects. Similarly, *DIFWriter* is the base class to generate DIF specifications from DIFGraph objects, and graph writer classes are only different in generating model specific built-in attributes and model keywords. Therefore, all specialized classes are extended from the base classes, and only a small set of model-specific methods are overridden or added.

4.4.3 Algorithm Implementation

For supported dataflow models, the DIF package also provides efficient implementations of various useful analysis, scheduling, and optimization algorithms in Java that operate on the DIF representations (dataflow graph objects). Algorithms currently available in the DIF package are mainly in SDF and its closely related models, and they are based primarily on well-developed algorithms such as repetitions vector computation, consistency validation, buffer minimization, and scheduling [7, 45, 60].

The dataflow-based algorithms in the DIF package provide designers an efficient programming interface to analyze and optimize DSP applications. By building on the DIF representations and existing algorithms, emerging techniques can be developed easily in the DIF package. It is also worthwhile to integrate DSP design tools with the DIF package and then utilize the powerful scheduling and optimiza-

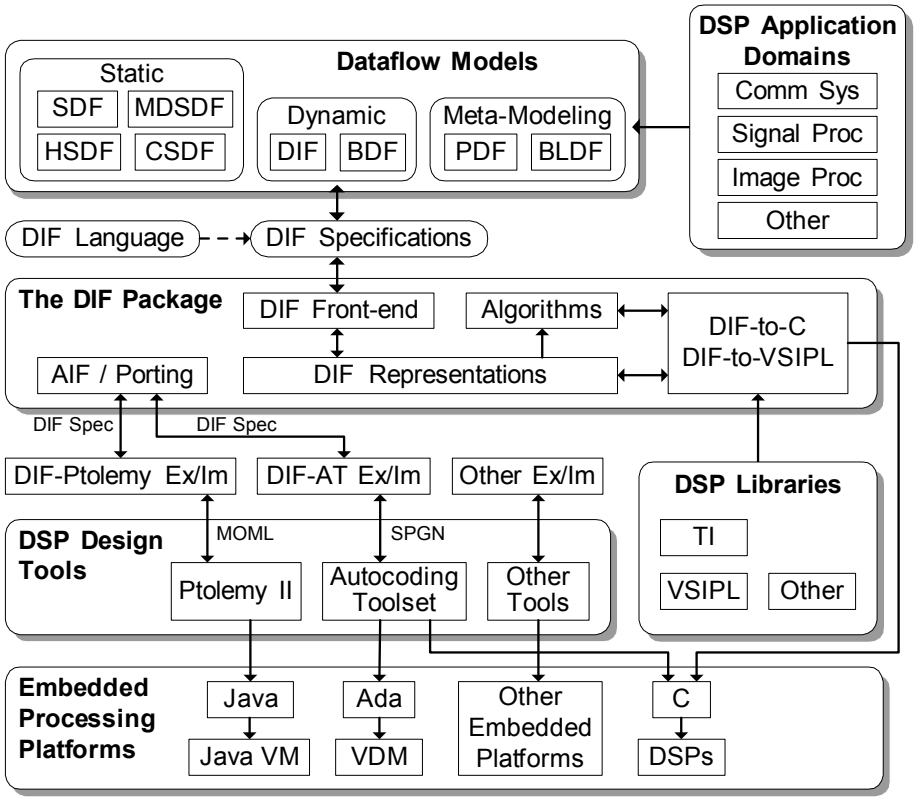


Figure 4.11: The role of DIF in DSP system design.

tion features in the DIF package.

4.5 The Methodology of Using DIF

In the previous sections, we have presented the overall DIF framework including the DIF language, DIF package, DIF-based porting methodology, and DIF-to-C software synthesis framework. Here, we introduce a general approach to using the DIF framework in dataflow-based DSP system design. Figure 4.11 illustrates the end-user viewpoint of the DIF framework. DIF supports a layered design methodology covering: 1) DSP application domains, 2) dataflow models, 3) the DIF package, 4) DSP design tools, 5) DSP libraries, and 6) embedded processing platforms.

In general, our target application domain is the broad domain of digital signal processing, including applications for processing of signals associated with digital communications, and with audio, image, video, and multimedia data streams. Dataflow models of computation have been shown very useful in modeling applications in this general domain (e.g., see [51, 7, 44, 37, 74]. Specific forms of dataflow that are relevant to DSP system design include 1) static dataflow models such as synchronous dataflow [51], cyclo-static dataflow [10], homogeneous synchronous dataflow [51, 75], multi-dimensional synchronous dataflow [62], windowed synchronous dataflow [40], and scalable synchronous dataflow (SSDF) [69]; 2) dynamic dataflow models such as Boolean-controlled dataflow (BDF) [11], well-behaved dataflow [24], reactive process networks [25], Compaan process networks [19], and the general DIF model (see Section 4.3); and 3) meta-modeling techniques such as parameterized dataflow (PDF) [3] and blocked dataflow (BLDF) [44]. Many of the above dataflow models are currently supported in DIF or under investigation for future incorporation into DIF.

The primary DSP design tools that we have been experimenting with in our development of DIF so far are the SDF domain of Ptolemy II [21], developed at UC Berkeley; the Autocoding Toolset developed by MCCI [70]; the ADS tool from Agilent Technologies [67]; LabVIEW from National Instruments [2]; and Compaan from Leiden University [76]. However, DIF is in no way designed to be specific to these tools; our work with these tools is only as a starting point for experimenting with DIF in conjunction with sophisticated academic and industrial DSP design tools.

The embedded processing platforms layer in Figure 4.11 gives examples of platforms supported by Ptolemy II, the Autocoding Toolset, and Texas Instruments (TI) DSP libraries. Ptolemy II runs on Java; the Autocoding toolset is able to generate C code for Mercury DSPs and Ada for the Virtual Design Machine (VDM) [55]; and TI DSP libraries are optimized for TI DSPs. In general, this layer represents all embedded processing platforms that are supported by DSP design tools and DSP libraries.

The DIF language and the DIF package provide an intermediate layer between abstract dataflow models and various practical implementations. DIF provides users an integrated programming interface to work with different layers in Figure 4.11. Using the DIF language, DSP applications modeled in various dataflow semantics can be specified as textual DIF specifications, and then realized in DIF representations (dataflow graph objects) through the DIF front-end interface. Alternatively, users can also construct DIF representations directly by using the DIF dataflow graph library. Once they are working with DIF specifications, users can then utilize various dataflow-based algorithms provided in the DIF package to analyze, schedule, and optimize their DSP applications. The extensibility of the DIF package also benefits users in developing new dataflow models and algorithms — emerging techniques can be implemented easily by building on or extending the existing dataflow graph classes and algorithms.

As discussed in Chapter 1, integrating complementary capabilities and platform support capabilities from different design tools and libraries is an important objective for DSP system designers. With the novel DIF-based porting methodol-

ogy that uses DIF as an intermediate format and AIF for specifying actor mapping information, migrating or developing DSP designs across multiple tools and libraries can be done systematically. This approach indeed relies on the support of DIF from tools (i.e., exporting and importing capabilities). Building support between DIF and design tools can also provide the DSP design industry a useful front-end to use DIF and the DIF package, e.g., utilizing the powerful scheduling and optimization features in the DIF package.

In addition to the synthesis capabilities provided by design tools, the DIF-to-C software synthesis framework in conjunction with off-the-shelf DSP libraries provides users a new path to software implementations from standalone use of the DIF package. With the novel DIF-to-C framework, DIF users can easily utilize different DSP library functions, integrate their own actor implementations with dataflow models, and explore performance trade-offs through various dataflow techniques. By integrating software synthesis with our DIF-based porting methodology, users can further explore design and implementation choices (tools, libraries, platforms) through the DIF framework.

Chapter 5

DIF-Based Porting Methodology

In this chapter, we present the DIF-based porting methodology for systematically porting DSP applications across design tools and libraries. Our porting methodology integrates DIF tightly with the specific exporting and importing mechanisms which interface DIF to specific DSP design tools. In conjunction with this porting mechanism, we present a novel language, called *actor interchange format* (AIF), for transferring relevant information pertaining to DSP library components across different tools. Through a case study of a synthetic aperture radar (SAR) application, we demonstrate the efficiency and the high degree of automation offered by our DIF-based porting approach.

5.1 Exporting and Importing

In DIF terminology, *exporting* means translating a DSP application from a tool's specification format to DIF (either to the DIF language or directly to the appropriate form of DIF representation). On the other hand, *importing* means translating a DIF specification or a DIF representation to a design tool's specification format or its internal representations. In general, exporting and importing processes are tool-dependent. Directly parsing and translating between DIF and tools' specification formats is usually inefficient.

We develop a new exporting and importing approach based on *dataflow graph mapping*. Our general approach for exporting is to comprehensively traverse graphical representations in a design tool and then map the encountered components into corresponding components or equivalent groups of components that are available in DIF dataflow graph library; and similarly, the approach for importing is done in the reverse manner. Dataflow-based design tools usually have their own specific representations instead of just the abstract components defined in theoretical dataflow models. However, since DIF provides 1) a complete set of object-oriented classes (DIF dataflow graph library) for representing dataflow graphs and 2) a front-end interface (DIF front-end) for converting between the representations and the DIF language, traversing and mapping between the graphical (internal) representations of tools and the formal dataflow representations in DIF is feasible and is typically more efficient to develop and execute.

Even though DIF is developed in Java and may not directly be used by C/C++ based design tools, through our new development [17] of C/C++ library interfaces to DIF via Java Native Interface (JNI) [54] along with a wrapper generator system (JACE), our approach is now feasible in both Java and C/C++ environments.

Specifying an actor's computation and all necessary operational information is referred to as *actor specification*. Although this detailed information is not directly used by many dataflow-based analyses, it is essential in porting across tools and in hardware/software synthesis since every actor's functionality must be fully preserved. The actor block in the DIF language is designed for the actor specification. Lets take the FFT operation as an example to illustrate actor specification in DIF.

```
actor nodeID {
    computation = "ptolemy.domains.sdf.lib.FFT";
    order = intValue or intParamID;
    input = incomingEdgeID;
    output = outgoingEdgeID;
}
```

Figure 5.1: DIF actor specification of an FFT actor.

In Ptolemy II, the FFT actor is referred to as *ptolemy.domains.sdf.lib.FFT*, and it has a parameter *order* and two ports, *input* and *output*. The corresponding DIF actor specification is presented in Figure 5.1.

5.2 Porting Mechanism

The DIF-based porting mechanism consists of three major steps: 1) *exporting* — exporting a design from a tool to a DIF specification through a tool-specific DIF exporter, 2) *actor mapping* — mapping attributes of the original actors in the DIF specification to attributes associated with the corresponding target actors based on the given actor mapping information specified by the *actor interchange format* (AIF), and 3) *importing* — importing the mapped DIF specification to the target tool through a tool-specific DIF importer.

The porting mechanism illustrated in Figure 5.2 is based on an experiment of porting from the Autocoding Toolset [70] to Ptolemy II [21]. The first step is to export a DSP application developed in the Autocoding Toolset (AT), which uses MCCI's Signal Processing Graph Notation (SPGN) as its specification format, to the corresponding DIF specification through the DIF-AT exporter developed by MCCI. In this DIF specification, actor information is specified for the Autocoding Toolset.

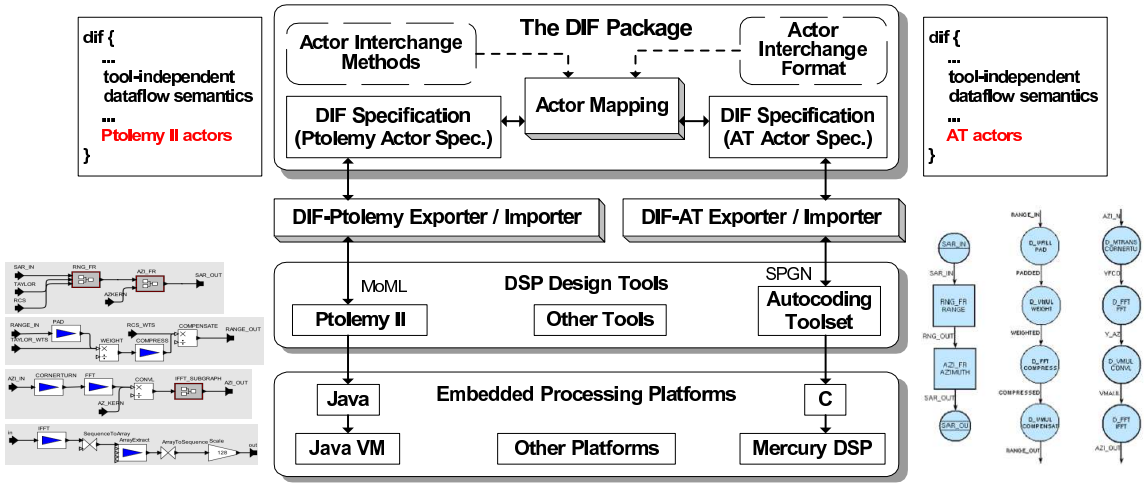


Figure 5.2: The DIF-based porting mechanism.

In the second step, actor mapping mechanism interchanges the tool-dependent actor information from the Autocoding Toolset to Ptolemy II in the DIF specification. DIF is used as an intermediate state in the porting process. The final step is to import the DIF specification with actor information specified for Ptolemy II to the corresponding Ptolemy II graphical representation and then to an equivalent Ptolemy II Modeling Markup Language (MoML) [52] format. This importing process is handled by the DIF-Ptolemy exporter/importer developed in this work.

The key idea behind the DIF-based porting approach is that except for actor information, a DIF specification for a DSP application represents the same dataflow semantics regardless of which design tool is used to generate it, and furthermore, porting DSP applications can be achieved by properly mapping the tool-dependent actors, while transferring the dataflow semantics unaltered.

5.3 Actor Mapping

The objective of *actor mapping* is to map an actor in a design tool to an actor or a set of actors in another design tool while preserving the same functionality. Because different design tools generally provide different sets of actor libraries, problems may arise due to actor absence, actor mismatch, and actor attribute mismatch.

If a design tool does not provide the corresponding actor, the *actor absence* problem arises. If corresponding actors exist in both libraries but the specific functionalities of those actors do not completely match, we encounter the *actor mismatch* problem. For example, the FFT domain primitive (library function) in the Autocoding Toolset allows users to indicate an FFT or IFFT operation through its parameter *FI*, but the FFT actor in Ptolemy II does not. *Actor attribute mismatch* arises when attributes are mapped between actors but the values of corresponding attributes cannot be directly interchanged. For example, the parameter *order* of the Ptolemy FFT actor specifies the FFT order, but the corresponding parameter N of the Autocoding Toolset FFT domain primitive specifies the length of FFT.

We develop the *actor interchange format* (AIF) for specifying how to map actors (i.e., actor-to-actor mapping and actor-to-subgraph mapping) across pairs of tools. AIF can significantly ease the burden of actor mismatch problems by allowing designers to specify how multiple actors in the target design tool can construct a subgraph such that the subgraph's functionality is compatible with the source actor. Such conversions reduce the need for users to introduce new actor definitions in the target tool.

We also develop the *actor interchange methods* that can be optionally given in AIF specifications to perform conditional checks or to evaluate attribute values. Actor interchange methods can solve attribute mismatch problems by evaluating a target attribute in a consistent, centrally-specified manner, based on any subset of source attribute values.

For absent actors, most design tools provide ways to create actors through some sort of actor definition language. Once users determine equivalent counterparts for absent actors and specify the mapping information in AIF properly, DIF porting mechanism can take over the job efficiently and systematically.

5.3.1 Actor Interchange Format

Actor interchange format (AIF) is a specification format dedicated to actor mapping. It provides syntax to specify actor interchange information, including: 1) mapping from a source actor (an actor in the source design tool) to a target actor (an actor in the target design tool), 2) mapping from a source actor to a subgraph consisting of a set of target actors, 3) mapping from source attributes to target attributes, and 4) optionally specifying the prior condition to trigger a mapping as well as the method and expression to determine an attribute value. We present AIF syntax partially in Figure 5.3 and Figure 5.4, where items in boldface are built-in keywords; non-bold items are specified by users or generated by tools; items enclosed by squares are optional; and "..." represents optionally repeated statements.

The actor-to-actor mapping syntax, as presented partially in Figure 5.3, spec-

```
actor targetActor <− sourceActor [ | methodID(arg, ..., arg) ] {
    targetAttributeID = value;
    targetAttributeID <− sourceAttributeID [ | methodID(arg, ..., arg) ];
    ...
}
```

Figure 5.3: The AIF actor-to-actor mapping syntax.

ifies the mapping information from a source actor *sourceActor* to a target actor *targetActor*. A method *methodID* is given optionally to specify a prior condition that must be satisfied to trigger the mapping. AIF allows users to directly assign a value *value* for a target attribute *targetAttributeID*. In addition, the value of *targetAttributeID* can also be directly assigned by the value of a source attribute *sourceAttributeID*, or a method *methodID* can be given optionally to evaluate the value of *targetAttributeID* based on the runtime values of source actor attributes.

The actor-to-subgraph mapping syntax, as presented partially in Figure 5.4, specifies the mapping from a source actor *sourceActor* to a subgraph *targetGraph* consisting of a set of target actors. It is designed for use when matching to a standalone actor in the target tool is not possible. The topology block portrays the topology of *targetGraph*. The interface block defines the input and output ports of *targetGraph*, and also specifies mappings from the interface attributes *sourceAttributeID* of *sourceActor* to the interface ports *portID* of *targetGraph*. The actor information of each node in *targetGraph* is specified in a separate actor block, where the syntax is pretty much the combination of the DIF actor block and the AIF actor-to-actor mapping block.

AIF grammar and AIF parser are developed based on SableCC [22]. For more detailed information about AIF, we refer the reader to [31].

```
graph targetGraph <− sourceActor [ | methodID(arg, ..., arg) ] {
    topology {
        nodes = nodeID, ..., nodeID;
        edges = edgeID (sourceNodeID, sinkNodeID), ..., edgeID (sourceNodeID, sinkNodeID);
    }
    interface {
        inputs = portID [: nodeID] <− sourceAttributeID, ...,
                 portID [: nodeID] <− sourceAttributeID;
        outputs = portID [: nodeID] <− sourceAttributeID, ...,
                  portID [: nodeID] <− sourceAttributeID;
    }
    actor nodeID {
        computation = "stringDescription";
        attributeID = value;
        attributeID = ID;
        attributeID = ID, ..., ID;
        targetAttributeID <− sourceAttributeID [ | methodID(arg, ..., arg) ];
        ...
    }
}
```

Figure 5.4: The AIF actor-to-subgraph mapping syntax.

5.3.2 Actor Interchange Methods

The methods optionally given in AIF specifications are referred to as *actor interchange methods*. A set of commonly-used interchange methods is defined in a built-in Java class in the DIF package. Users can extend this class and design specific interchange methods for more complicated or specialized actor mapping scenarios.

There are three built-in actor interchange methods in the DIF package: 1) *ifExpression*("*expression*") evaluates the Boolean *expression* and returns true or false; 2) *assign*("*expression*") evaluates *expression* and returns the evaluated value; and 3) *conditionalAssign*("*valueExpression*", "*conditionalExpression*") returns the value of *valueExpression* if the *conditionalExpression* is true. Note that the attributes of the source actor can be used as variables in expressions and their values are used at runtime during evaluation.

```
actor ptolemy.domains.sdf.lib.FFT <- D_FFT | ifExpression("FI == 0") {
    order <- N | conditionalAssign("log(N)/log(2)","(log(N)/log(2))-rint(log(N)/log(2))==0");
    input <- X;
    output <- Y;
}
```

Figure 5.5: AIF specification for mapping FFT.

5.3.3 Case Study: FFT

According to the actor mismatch and attribute mismatch problems described in Section 5.3, the Autocoding Toolset FFT domain primitive (which is referred to as *D_FFT* in MCCI domain primitive library) can be mapped to the Ptolemy FFT actor only when its parameter *FI* is not set to indicate IFFT operation. Moreover, the parameter N of D_FFT can be mapped to the parameter *order* of Ptolemy's FFT actor only when $N = 2^{order}$ is satisfied. The AIF specification for mapping the FFT operation from the Autocoding Toolset to Ptolemy II is partially shown in Figure 5.5.

The D_FFT domain primitive also has a parameter B, which specifies the first point of its output sequence, and a parameter M, which specifies the number of output points. Furthermore, there is a factor of N difference between the IFFT operation of D_FFT and the Ptolemy IFFT actor. One way to solve this problem is to create a new IFFT actor in Ptolemy, but this approach is relatively time-consuming. The actor-to-subgraph mapping feature in DIF can be used as a more convenient alternative. Figure 5.6 presents the critical part of this AIF mapping specification. If a D_FFT domain primitive indicates an IFFT operation ($FI == 1$) and it outputs only part of its sequence ($M \neq N$), it is mapped to a Ptolemy

```
graph ptolemy.actor.TypedCompositeActor <- D_FFT | ifExpression("FI==1 && M!=N") {
    topology {
        nodes = IFFT, Scale, SequenceToArray, ArrayExtract,ArrayToSequence;
        edges = ...;
    }
    interface {
        inputs = in : IFFT <- X;
        outputs = out : ArrayToSequence <- Y;
    }
    actor IFFT {
        computation = "ptolemy.domains.sdf.lib.IFFT";
        order <- N | conditionalAssign("log(N)/log(2)",
                "(log(N)/log(2))-rint(log(N)/log(2))==0");
        ...
    }
    actor Scale {
        ...
        factor <- N;
    }
    actor ArrayExtract {
        ...
        sourcePosition <- B | assign("B-1");
        extractLength <- M;
    }
    ...
}
```

Figure 5.6: AIF specification for mapping IFFT.

subgraph consisting of an IFFT actor for performing an IFFT operation, a Scale actor for adjusting each sample by a factor of N, and three array processing actors for extracting a certain part of the output sequence. The input and output ports of the subgraph, *in* and *out*, are mapped from parameters X and Y of D_FFT. For complete AIF specification of this mapping, we refer the reader to [31].

5.4 Experiment

In the experiment, we port a synthetic aperture radar (SAR) benchmark application from the Autocoding Toolset [70] to Ptolemy II [21]. Figure 5.7 shows the SAR system developed in Autocoding Toolset. Figure 5.7.(a) illustrates the

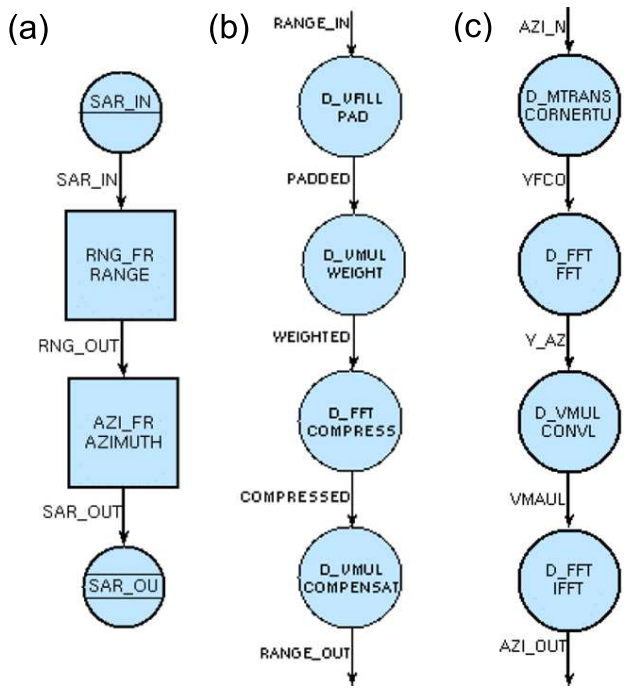


Figure 5.7: The SAR system in the Autocoding Toolset.

top-level dataflow graph, which consists of two major building blocks: RANGE processing in Figure 5.7.(b) and AZIMUTH processing in Figure 5.7.(c). With a properly-designed actor interchange specification together with actor interchange methods available in the DIF package [31], the DIF actor mapping mechanism can translate the DIF specification of Figure 5.7, which is exported from the Autocoding Toolset, to an equivalent DIF specification for Ptolemy II. The DIF-Ptolemy importer then imports this equivalent specification, and the resulting SAR application in Ptolemy II is shown in Figure 5.8. Figure 5.8.(a), (b), and (c) correspond to Figure 5.7.(a), (b), and (c), respectively. Note that the mismatched actor IFFT in Figure 5.7.(c) is mapped to the IFFT_SUBGRAPH in Figure 5.8.(d) through the AIF actor-to-subgraph mapping capability.

The ported SAR benchmark application in Ptolemy II works correctly. Figure

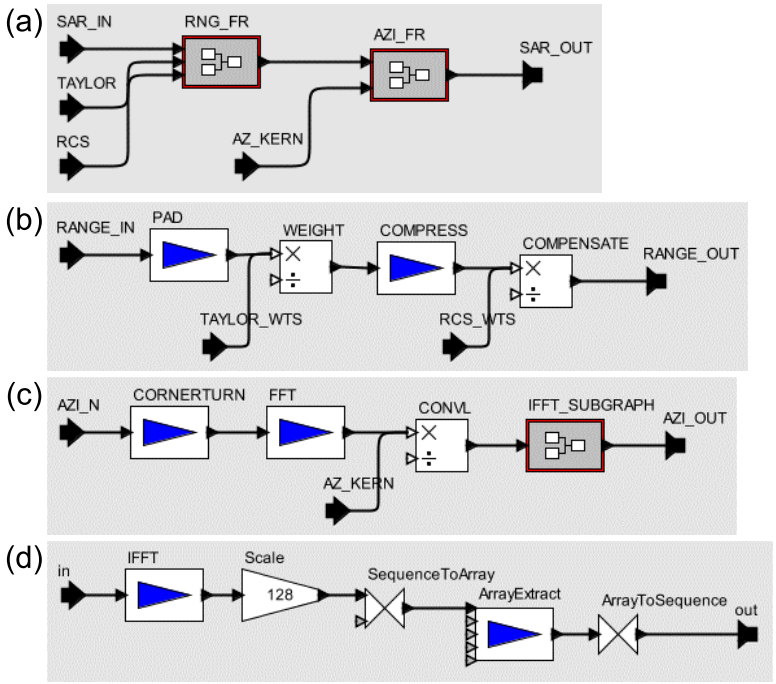


Figure 5.8: The ported SAR system in Ptolemy II.

Ptolemy II

```
1.113328370E9, -5.672582199E8
1.686243152E9, -1.132239286E9
2.280892492E9, -1.837179778E9
2.787030647E9, -2.565079199E9
3.121469726E9, -3.124321013E9
3.235633491E9, -3.339997173E9
3.126105298E9, -3.132702116E9
2.795907223E9, -2.578937710E9
2.292518065E9, -1.852489499E9
1.698661416E9, -1.145532955E9
```

Autocoding Toolset

```
1.11334E+09, -5.67194E+08
1.68657E+09, -1.13206E+09
2.28101E+09, -1.83712E+09
2.78720E+09, -2.56485E+09
3.12169E+09, -3.12429E+09
3.23570E+09, -3.33972E+09
3.12633E+09, -3.13268E+09
2.79604E+09, -2.57867E+09
2.29266E+09, -1.85242E+09
1.69888E+09, -1.14531E+09
```

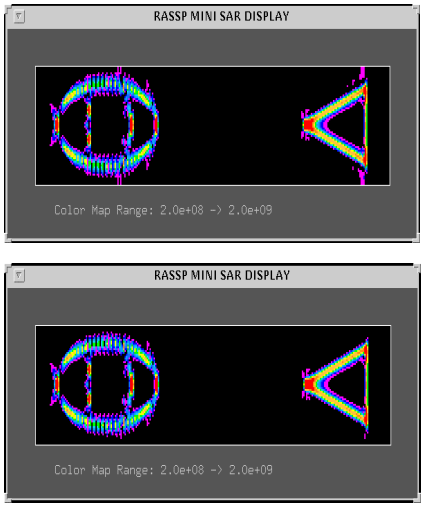


Figure 5.9: SAR simulation results in Ptolemy II and the Autocoding Toolset.

5.9 compares the output samples generated by both tools. The simulation results are the same except for tolerable precision errors.

Chapter 6

DIF-to-C Software Synthesis

In this chapter, we present our *DIF-to-C software synthesis framework* for automatically generating C-code implementations from high-level dataflow modeling of DSP systems that are programmed in DIF. Comparing to general software synthesis tools, the DIF-to-C framework possesses the following unique features:

1. *Library-neutral*: In contrast to built-in actor libraries in conventional EDA tools, our software synthesis framework is library-neutral such that DIF programmers can associate actors with desired C functions either designed by themselves or obtained from existing libraries. The DIF-to-C framework currently supports general C-based libraries, e.g., DSP libraries from Texas Instruments [80, 79], and can be easily extended to support more specialized C-based APIs, such as VSIPL [39].

2. *Design space exploration*: The DIF package provides representations of various dataflow models and efficient implementations for many scheduling algorithms and buffering techniques. This large and growing set of models, algorithms, and techniques spans a broad range of the design space: designers can easily explore different combinations and determine trade-offs among key metrics such as code size, memory requirements, and performance.

3. *Portability*: By integrating the DIF-to-C framework with the systematic DIF-based porting approach (see Chapter 5), a DIF specification of a design can be ported and synthesized on various embedded processing platforms.

The DIF-to-C software synthesis framework is presently based on SDF semantics. Figure 6.1 illustrates the design flow that underlies the DIF-to-C framework. In the programming phase, we model a DSP application using SDF, and specify the modeling information in DIF, including graph topologies, hierarchical structures, dataflow behavior (production rates, consumption rates, and delays), actor attributes (actor-function associations, edge/port connections, parameters, etc.), and all other relevant information (e.g., data types). In particular, actors in the DIF specification are specified based on the chosen C functions. Next, we use the DIF front-end interface to construct the internal DIF representation, i.e., the dataflow graph objects realizing the DIF specification. This object representation is then passed as input to the subsequent scheduling, buffering, and code generation techniques.

In the rest of this chapter, we introduce the novel developments in the scheduling and code generation phases and present the simulation results of several synthesized DSP applications.

6.1 Scheduling

In the scheduling phase, we compute a *schedule* of the SDF graph through one of various scheduling algorithms. By a *schedule*, we mean a sequence of actor

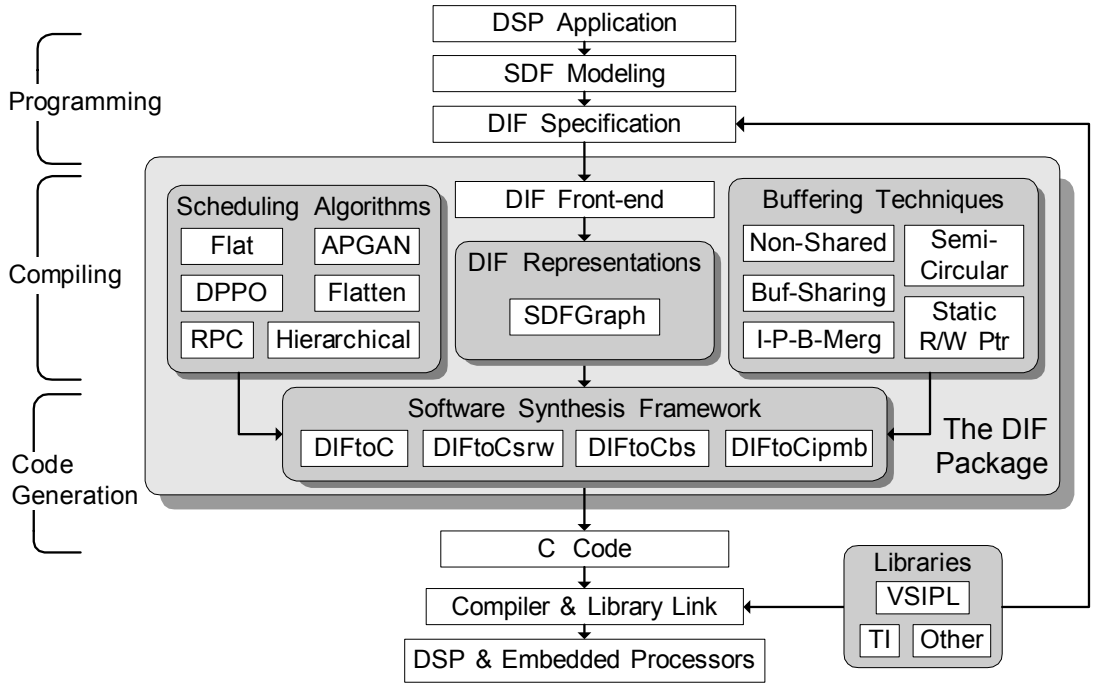


Figure 6.1: DIF-to-C software synthesis framework.

firings or more generally, any static, dynamic, or hybrid static/dynamic sequencing mechanism for executing actors. The DIF-to-C framework is mainly based on SDF semantics. As a result, we focus on purely static scheduling, which is most natural for SDF graphs. There is a complex range of trade-offs involved during the scheduling phase, and the DIF package provides a variety of scheduling algorithms and strategies for exploring trade-offs.

6.1.1 SDF Scheduling Preliminaries

As reviewed in Section 2.1.1, an SDF graph $G = (V, E)$ has a valid schedule (is *consistent*) if it is free from deadlock and is sample rate consistent — i.e., if there

is a positive integer solution to the *balance equations*:

$$\forall e \in E,\ prd(e) \times \boldsymbol{x}[src(e)] = cns(e) \times \boldsymbol{x}[snk(e)]\ . \tag{6.1}$$

The minimum positive integer solution $\boldsymbol{q}_G$ for the vector $\boldsymbol{x}$ is called the *repetitions vector* of G. A *valid minimal periodic schedule* is then a sequence of actor firings in which each actor v is fired for its *repetition count* $\boldsymbol{q}_G[v]$ times, and the firing sequence obeys the data-driven properties imposed by the SDF graph.

Based on Section 2.1.2, given a schedule S, we define the *buffer size* required for an edge e, $buf(e)$, to be the maximum number of tokens simultaneously queued on e during an execution of S, $maxToken(e, S)$; and the *total buffer requirement* of an SDF graph $G = (V, E)$ to be the sum of the buffer sizes of all edges:

$$buf(G) = \sum_{\forall e \in E} maxToken(e, S)\ . \tag{6.2}$$

As discussed in Section 2.1.1, actor firing sequences can be represented through looping constructs [7] for memory-efficient storage. A *schedule loop*, $L = (n\ T_1 T_2 \cdots T_m)$, is defined as the successive repetition n times of the invocation sequence $T_1 T_2 \cdots T_m$, where each T_i is either an actor firing or a (nested) schedule loop. A looped schedule $S = L_1 L_2 \cdots L_N$ is an SDF schedule that is expressed in terms of the schedule loop notation. If every actor appears only once in S, S is called a *single appearance schedule* (SAS), otherwise, S is called a *multiple appearance schedule* (MAS).

Any SAS for an acyclic SDF graph can be represented in the *R-schedule* form [7], which can be naturally represented as a *schedule tree*. A schedule tree is in turn a binary tree where an internal node represents a sub-schedule and a leaf node

represents an actor firing. It provides a convenient internal representation for SDF scheduling, and is widely used in computing schedules and buffer minimization. An example of an R-schedule and the corresponding schedule tree is shown in Figure 6.3.

6.1.2 Scheduling Algorithms

The thorough review of SDF scheduling algorithms is provided in Chapter 3. In general, the problem of computing a buffer-optimal SDF schedule is NP-complete, and buffer-optimal schedules are usually MASs whose lengths generally increase exponentially in the size of the SDF graph. An SAS is often preferable in software synthesis due to its optimally compact implementation containing only a single copy of code for every actor. A valid SAS exists for any consistent and acyclic SDF graph and can be easily derived from a *flat scheduling* strategy, i.e., a strategy that computes a topological sort of an SDF graph G and iterates each actor v $\boldsymbol{q}_G[v]$ times. However, flat scheduling may also lead to relatively large buffer requirements and latencies in multirate systems [7].

For joint code and data minimization in software synthesis, several scheduling algorithms have been developed in acyclic SDF graphs. The *acyclic pairwise grouping of adjacent nodes* (APGAN) [7] technique is a heuristic to generate a buffer-efficient topological sort (and looped schedule). The *dynamic programming post optimization* (DPPO) [7] performs dynamic programming over a given actor ordering (topological sort) to generate a buffer-efficient looped schedule. It has sev-

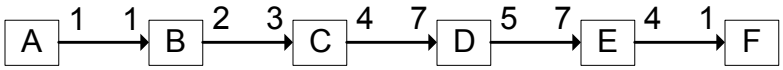


Figure 6.2: A CD-DAT SDF graph.

eral forms for different cost functions, e.g., GDPPO [7], CDPPO [85], and SDPPO [59].

For a graph containing cycles, an SAS may or may not exist depending on whether the numbers and locations of delays in its cycles satisfy certain sufficiency conditions. The *loose interdependence algorithm framework* (LIAF) [7] has been developed for generating SASs whenever they exists. Beyond SASs, the work of [45] presents a recursive procedure call (RPC) based technique that generates MASs from a given R-schedule through recursive graph decomposition. The resulting procedural implementation is proven to be bounded polynomially in the graph size. This MAS technique significantly reduces memory requirement over SAS at the expense of some moderate runtime overhead.

The aforementioned algorithms are implemented in the DIF package and integrated in the DIF-to-C framework. Figure 6.2 shows an SDF graph of a multi-rate CD-to-DAT sampling rate conversion system. Table 6.1 presents schedules computed from various SDF scheduling algorithms and their corresponding buffer requirements.

Table 6.1: Schedules and buffer requirements.

Algorithm	Schedule	Buffer
Flat	(147A)(147B)(98C)(56D)(40E)(160F)	1273
APGAN	(49(3AB)(2C))(8(7D)(5E(4F)))	438
DPPO	(7(7(3AB)(2C))(8D))(40E(4F))	347
RPC-based MAS	(2(1(1(7(1(1AB)(2(AB)C))D)D)(5E(4F)))(2(1(1(7(1(1AB)(2(1AB)C))D)D)(5E(4F)))(1E(4F))))(1(1(1(7(1(1AB)(2(1AB)C))D)D)(5E(4F)))(1E(4F)))	69

6.1.3 Scheduling Hierarchical SDF Graphs

The aforementioned scheduling algorithms are designed for scheduling flattened SDF graphs. As a useful alternative to this form of scheduling, a *hierarchical scheduling strategy* is developed in the DIF-to-C framework. In hierarchical scheduling, the original hierarchical structure (i.e., the *design hierarchy*) is preserved in the generated code. Specifically, in our approach, each hierarchical subsystem is instantiated as a separate subroutine. Hierarchical scheduling is desirable in cases where it is useful to maintain a correspondence between the design hierarchy and the structure of the generated code. For example, such a correspondence can be useful as a debugging aid, and it can also lower the complexity of scheduling. Our approach to hierarchical scheduling is primarily based on SDF clustering [7]. That is, our hierarchical scheduling approach operates by recursively scheduling all subgraphs using any given scheduling algorithms, and then updating the production and consumption rates of supernodes such that firing a supernode corresponds to executing one iteration of the minimum periodic schedule of the subgraph.

To accommodate situations in which designers do not need to impose the hierarchical scheduling constraint, we also provide a *flattened scheduling strategy* in our

framework. In this approach, all nested hierarchies are flattened before scheduling. This form of scheduling can in general lead to more efficient schedules (because the design space of permissible schedules is usually much larger); however, the schedules are much more difficult to understand in relation to the original SDF graph.

For the algorithm of both strategies, we refer the reader to [36].

6.2 Buffering

The last step in the scheduling phase is to allocate and manage buffers. Although edges in an SDF graph conceptually represent FIFO buffers, implementing a FIFO structure usually leads to severe runtime and memory overhead due to maintaining the strict FIFO operations. In the DIF-to-C framework, only the necessary amount of memory space is allocated for each edge, and buffers are managed between actor firings (i.e., function or subroutine calls) such that actor firings always access the correct subsets of live tokens. In this section, we present several buffering techniques that have been implemented in the DIF-to-C framework for exploring buffering trade-offs.

6.2.1 Buffer Allocation

The total buffer requirement defined in Equation (2.2) is based on the *non-shared memory model*, i.e., each buffer is allocated individually in memory and is live throughout a schedule. In fact, the scheduling algorithms described above are developed for improving memory requirements based on this model. Given a

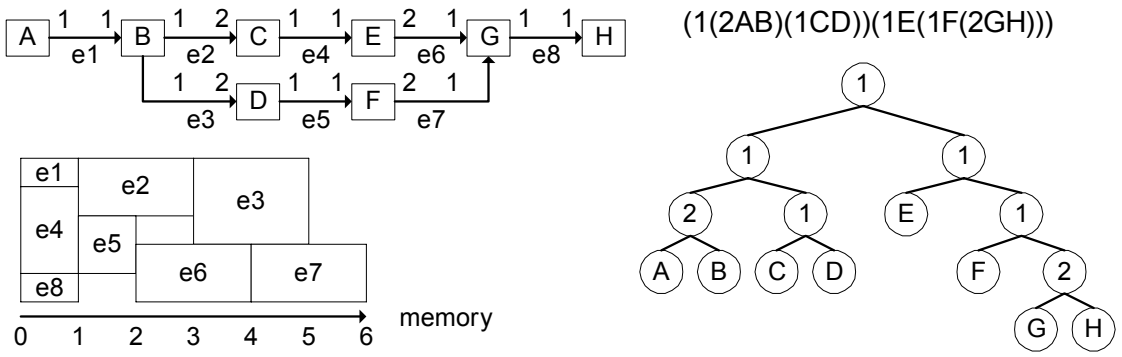


Figure 6.3: A buffer sharing example.

schedule, the *non-shared buffering* technique simply allocates a buffer (declares an array) for each edge independently.

In practice, memory space can be reduced by sharing memory across multiple buffers as long as their lifetimes (at the granularity of actor firings) do not overlap, and a systematic *buffer sharing* technique has been developed in [59] based on this motivation. In this technique, an R-schedule is first computed through SDPPO [59], and then a schedule tree is constructed to efficiently extract lifetime parameters. Next, the first-fit heuristic is applied to pack arrays efficiently into memory and determine the actual memory requirement and the buffer (array) locations. Figure 6.3 presents a simple example for illustrating this technique, and for a complete derivation, we refer the reader to [59].

As discussed in Chapter 3, the concept of buffer merging is developed formally in [6]. Certain DSP computations can be executed *in-place* such that a single buffer is sufficient for both input and output edges, e.g., the discrete cosine transform (IMG_fdct_8x8) in the Texas Instruments DSP library [79]. An in-place actor is naturally suitable for merging its input and output edges, and buffer merging may be

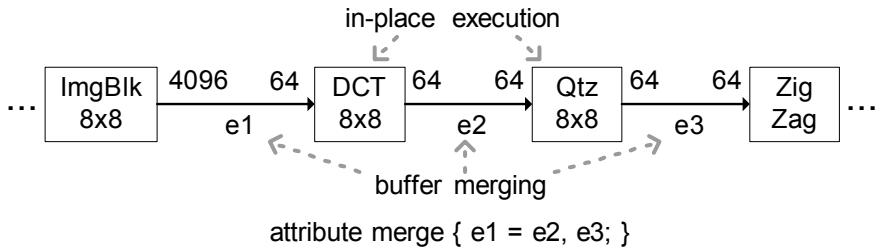


Figure 6.4: An in-place buffer merging example in JPEG.

strictly required if the in-place actor is invoked through a pre-defined function that has only one argument for both input and output. We have developed the *in-place buffer merging* technique to merge buffers for an in-place actor or a sequence of in-place actors. In dataflow modeling, it is not natural to represent a single merged edge for an in-place actor, because dataflow edges also impose precedence constraints. For this reason, an edge attribute *merge* is dedicated in DIF to specify exactly where in-place buffer merging takes place. Figure 6.4 presents a sequence of in-place actors in a JPEG subsystem and the corresponding buffer merging specification in DIF.

A sequence of edges $e_1, e_2, \ldots, e_N$ in an SDF graph G can be merged for in-place execution if 1) they are connected, i.e., $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, ..., $snk(e_{N-1}) = src(e_N)$, 2) the production rate and consumption rate of each in-place actor are the same, i.e., $cns(e_1) = prd(e_2)$, $cns(e_2) = prd(e_3)$, ..., $cns(e_{N-1}) = prd(e_N)$, and 3) the edges are delayless. In our approach, we allocate (declare) only a single buffer (array) for an edge e_i and merge (assign) others to it. Given a schedule S, e_i is chosen such that the least common ancestor of $src(e_i)$ and $snk(e_i)$ is the highest internal node in the schedule tree of S, and this guarantees $maxToken(e_i, S)$ to be the maximum among $e_1, e_2, \ldots, e_N$, and in turn

prevents from buffer overflow.

6.2.2 Buffer Management

Knowledge of just the buffer size, $buf(e)$, and the buffer (array) address, $add(e)$, is not enough for actors to access the right place in the buffer at a particular iteration. Buffer management through *circular buffer* technique has been developed in [4]. In the DIF-to-C framework, inputs and outputs of actors (C functions) are passed by pointers through function arguments. This is a widely used convention in implementing DSP library functions, e.g., see [80, 79], and this convention generally assumes that input and output data are consecutive in memory space. However, this assumption prevents us from directly applying the circular buffer approach, since a particular firing may access tokens that circle around the buffer.

In the DIF-to-C framework, we develop the *semi-circular buffer* approach such that circular buffer is preserved, and input (output) data can be consumed (produced) consecutively. Given a schedule S, a buffer (array) is initially allocated (declared) for an edge e with enlarged size, $buf(e) = maxToken(e, S) + \max(prd(e), cns(e)) - 1$, to accommodate circled-around tokens for the worst case situation. The read and write pointers, $rp(e)$ and $wp(e)$, are initialized as: $rp(e) = 0$ and $wp(e) = del(e) \bmod maxToken(e, S)$.

For each firing of $src(e)$, it writes to the buffer at $add(e) + wp(e)$, and for each firing of $snk(e)$, it reads from the buffer at $add(e) + rp(e)$. Before a firing of $snk(e)$,

if $rp(e) + cns(e) > maxToken(e)$, the first $rp(e) + cns(e) - maxToken(e)$ tokens are copied to the position after $maxToken(e, S) - 1$ for accessing circled-around tokens in a linear manner. Similarly, after a firing of $src(e)$, if $wp(e) + prd(e) > maxToken(e)$, $wp(e) + prd(e) - maxToken(e)$ tokens after the position $maxToken(e, S) - 1$ are copied to the front. In addition, $rp(e)$ is updated as $rp(e) = (rp(e) + cns(e)) \bmod maxToken(e, S)$, and $wp(e)$ is update as $wp(e) = (wp(e) + prd(e)) \bmod maxToken(e, S)$

This approach can support all kinds of schedules and arbitrary edge delays. However, it also introduces buffer overhead for consecutive access and runtime overhead due to modulo and memory copy operations. If the input graph is delayless and the given schedule is an SAS, we can derive that $maxToken(e, S)$ is sufficient for periodic firings without circled-around access, and read and write pointers can be statically reset without modulo operations. Since a broad range of DSP subsystems can be modeled as acyclic, delayless SDF graphs, and because SASs are usually preferable, we develop the *static read/write pointer resetting* technique in the DIF-to-C framework for improving runtime and memory performance.

Given a delayless graph G and an SAS S, an edge e is only live in the schedule loop L that corresponds to the least common ancestor of $src(e)$ and $snk(e)$ in the schedule tree of S, i.e., neither $src(e)$ nor $snk(e)$ appears beyond L in S. In addition, $maxToken(e, S)$ is equal to the total number of tokens exchanged between $src(e)$ and $snk(e)$ within L. Based on these observations, we allocate $buf(e) = maxToken(e, S)$ for e, reset $rp(e)$ and $wp(e)$ at the beginning of the loop L, and update them after each firing of $src(e)$ and $snk(e)$ without modulo operations,

i.e., $rp(e) = rp(e) + cns(e)$ and $wp(e) = wp(e) + prd(e)$.

In fact, the worst case buffer requirements for the semi-circular buffer approach can be improved by simulating the buffer access behaviors at compile-time and allocating the exact semi-circular buffer requirements. This approach can be easily implemented and integrated in our framework.

6.3 Code Generation

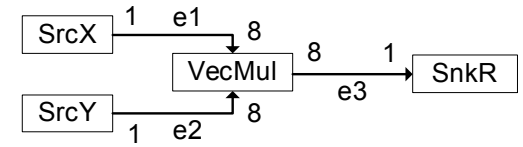
By integrating the DIF representations, scheduling algorithms, and buffering techniques, the code generation phase in the DIF-to-C framework is able to generate C implementations automatically. In this section, we describe our code generation algorithm and introduce how several strategies in this regard are developed in a systematic way. Finally, an executable is compiled from the generated code together with fine-grain actors (functions) or library links.

6.3.1 Function Prototype

Unlike general design tools that provide their own actor libraries, the DIF-to-C software synthesis framework is designed to support most C-based libraries. In order to support various C functions (actors), we impose the least-possible constraints: 1) input and output data should be passed by pointers through function arguments; and 2) the production and consumption rates should be fixed and known at compile time. Most C functions naturally conform these constraints.

Figure 6.5 illustrates the prototype of the vector multiplication function in the

void DSPF_sp_vecmul(float *x, float *y, float *r, int n)



actor VecMul {

computation = "DSPF_sp_vecmul";
x = e1;
y = e2;
r = e3;
n = 8;

}

Figure 6.5: Function prototype and actor specification.

Texas Instruments DSP Library [80]. The inputs x, y and output r are passed by "float*" pointers. The argument n indicates the number of elements in x, y, and r, which also implies that the production/consumption rate of x, y, and r is n. Figure 6.5 also shows an SDF example and the corresponding actor specification. Note that in current code generation approach, the order of actor attributes should preserve the order of arguments in the function prototype.

In practice, the data types of edges are required in code generation. The attribute *datatype* is used in the DIF-to-C framework for specifying the data type of edges, $type(e)$, and interface ports, $type(p)$, e.g., see Figure 4.3. In code generation, a buffer for e is declared as "$type(e)$ $e[size]$", where the buffer size is determined based on Section 6.2; when instantiating a subroutine for a subhierarchy, "$type(p) * p$" is generated as a subroutine argument for passing the buffer pointer from the outside connection.

6.3.2 DIFtoC Code Generator

In the DIF-to-C framework, *DIFtoC* is the base code generation class. It is developed based on the hierarchical scheduling strategy, non-shared buffer allocation, and the semi-circular buffer approach. In our code generation approach, a *main()* function is generated for the top-level hierarchy, and a subroutine is constructed for each sub-hierarchy recursively. For each loop in the schedule, a *for* loop construct is instantiated, and for each actor (or supernode) in the schedule, a function call (or a subroutine call) is instantiated. Edge buffers are declared as arrays, and code for managing circular buffers and updating read and write pointers is generated between function/subroutine invocations. The flattening scheduling strategy is also supported by flattening the top level hierarchy before scheduling. For the DIFtoC code generation algorithm in detail, we refer the reader to [36].

The DIFtoC code generation class schedules an SDF graph based on a properly given scheduling algorithm, and therefore provides flexibility in terms of scheduling. In our framework, integration with different buffering strategies can be implemented naturally by extending and overriding DIFtoC. Figure 6.6 presents the classes in the current DIF-to-C software synthesis framework. For SASs (in R-schedule form) and delayless graphs, we develop: 1) *DIFtoCsrw* that extends *DIFtoC* to implement static read/write pointer resetting, 2) *DIFtoCbs* that extends *DIFtoCsrw* to implement the buffer sharing technique, and 3) *DIFtoCipbm* that extends *DIFtoCsrw* to implement the in-place buffer merging technique. These code generation classes together with various scheduling algorithms in the DIF package provide a broad range

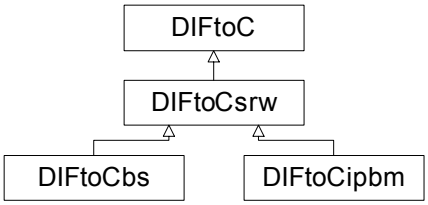


Figure 6.6: The class hierarchy in the DIF-to-C framework.

of design space.

The generated C code consists of two parts: The initialization part statically allocates buffers and declares read/write pointers and other parameters; The main body is mainly a looped sequence of function invocations (determined by the computed schedule) and interleaved with buffer management routines.

6.4 Experiment

The DSP applications in our DIF-to-C experiment include (a) CD-DAT and (b) DAT-CD sample rate conversion systems, (c) a four-level tree-structured filter bank, (d) a synthetic aperture radar (SAR) system, and (e) a JPEG encoder subsystem. We program the five coarse-grain SDF graphs in DIF, and then generate various C implementations based on different strategies through the DIF-to-C framework. Together with actor implementations either obtained from Texas Instruments signal and image processing libraries [80, 79] or manually implemented in C, we compile and simulate them in the Texas Instruments Code Composer Studio. The target simulation platform is the TMS320C64x DSP series and the compiler optimization setting is none.

Figure 6.7: DIF-to-C simulation results. (a) CD-DAT, (b) DAT-CD, (c) filter bank, (d) SAR, (e) JPEG.

Figure 6.7 presents the SDF graphs and the simulation results of the five applications. In our experiment, the combinations of scheduling and buffering strategies include (1) DIFtoC with the flat scheduling (C-F), (2) DIFtoC with APGAN-DPPO (C-AD), (3) DIFtoC with RPC-MAS (C-RPC), (4) DIFtoCsrw with APGAN-DPPO (SRW-AD), (5) DIFtoCbs with SDPPO [59] (BS-SD), and (6) DIFtoCipbm (for using in-place actors in the JPEG application) with APGAN-DPPO (IPBM-AD). Since the filter bank and SAR systems are modeled using hierarchical SDF graphs, we also present both the hierarchical and flattening scheduling results. Note that the actual possible combinations are much more than above.

The metrics we examined are: 1) memory (in bytes) — the total buffer memory space allocated for all dataflow edges, 2) code size (in bytes) — the compilation size of the generated C-code (including all of the automatically generated main function and subroutines, but excluding actor functions obtained from libraries or implemented by hand), 3) *CPU-excluded* (in cycles) — the cycles spent only in the generated code for one iteration of a minimum periodic schedule of the application SDF graph, and 4) *CPU-total* (in cycles) — the total CPU cycles for one iteration of the complete executable.

According to Figure 6.7, we found that there exists a complex range of trade-offs. For the CD-DAT and DAT-CD applications, RPC-based MAS significantly reduces memory requirements at the expense of code size. For the filter bank application, the buffer sharing method is an efficient approach, and the flattening strategy generally performs better than the hierarchical strategy. Even though the DIFtoC code generator allows MAS, it causes severe overhead in the SAR and JPEG appli-

cations. In these two cases, static read/write pointer resetting and buffer sharing can improve the situation significantly. For the JPEG application, since several operations can be executed in-place, the buffer-merging technique is very suitable. Regarding to the CPU-total metric for all applications, we found that the dataflow overhead (schedules, buffer allocation, and buffer management) is insignificant when taking large repetitions of heavily-computational actors into account. In general, such heavily computation-involved actors are usually optimized through compiler techniques or by hand.

6.5 Software Synthesis for MDSDF Graphs

MDSDF is introduced in Section 2.4. One of the problems in developing MDSDF-based software synthesis is that efficient mechanisms are required to rearrange data between MDSDF semantics and one-dimensional memory layouts.

The *vector, signal, and image processing library* (VSIPL) [39] is an open source, C-based API that provides various commonly used functions in vector and matrix computation, and many areas of signal processing. VSIPL adds a layer of abstraction involving the concepts of *blocks* and *views* to support portability across diverse memory and processor architectures. VSIPL blocks represent contiguous memory spaces where data is stored. VSIPL functions operate on views in a way that sets or subsets of data can be virtually arranged as vectors (1-D), matrices (2-D), or tensors (3-D). This feature makes VSIPL a particularly good match for integration with SDF and MDSDF semantics in software synthesis from DIF.

We have implemented the multi-dimensional dataflow representations, *MDSDFGraph*, and scheduling techniques in the DIF package. We have also developed *DIF-to-VSIPL* software synthesis capability [32] that supports both SDF and MDSDF by extending the original framework. Given the buffer space (1- or M-D) for a dataflow edge computed by scheduling and buffering techniques, the DIF-to-VSIPL code generation process creates a VSIPL block with size equal to the product of all dimensions. It also creates two VSIPL views (vector, matrix, or tensor views based on dimensions) associated with the block for source and sink actors (VSIPL functions). The *length* attributes of the views are decided by the production and consumption rates (1- or M-D), the *stride* attributes are determined by the buffer space (1- or M-D), and the *offset* attributes are adjusted between VSIPL functions based on the looped schedule (1- or M-D).

Figure 4.6 presents the input, output, and intermediate images computed by our synthesized C/VSIPL implementation of two-dimensional discrete wavelet transform (2DDWT).

This DIF-to-VSIPL capability augments the support of the DIF software synthesis framework to multiple useful dataflow models and extends the reach of DIF-based interchange to the wide variety of platforms that support VSIPL.

Chapter 7

Efficient Simulation of Critical Synchronous Dataflow Graphs

Synchronous dataflow (SDF) model of computation is widely used in EDA tools for system-level simulation. SDF representations of modern wireless communication systems typically result in *critical SDF graphs* — they consist of a large number of components and involve complex inter-component connections with highly *multirate* behavior. Simulating such systems using traditional SDF scheduling techniques generally leads to unacceptable simulation time and memory requirements on modern workstations and high-end PCs. In this chapter, we present a novel *simulation-oriented scheduler* (SOS) to provide effective, joint minimization of time and memory requirements for simulating critical SDF graphs. We have implemented SOS in the *Advanced Design System* (ADS) from Agilent Technologies. Our results from this scheduler demonstrate large improvements in simulating real-world, large-scale, and highly multirate wireless communication systems (e.g., 3GPP, Bluetooth, 802.16e, CDMA 2000, XM radio, EDGE, and Digital TV).

7.1 Introduction

SDF scheduling and buffering preliminaries are introduced in Section 2.1; A thorough review of SDF scheduling algorithms and buffering techniques is presented in Chapter 3.

Generally, the design space of SDF schedules is highly complex, and the schedule has a large impact on the performance and memory requirements of an implementation [7]. For synthesis of embedded hardware/software implementations, memory requirements (including memory requirements for buffers and for program code) are often of critical concern, while tolerance for compile time is relatively high [56], so high complexity algorithms can often be used. On the other hand, for system simulation, simulation time (including time for scheduling and execution) is the primary metric, while memory usage (including memory for buffering and for the schedule) must only be managed to fit the available memory resources.

Scheduling in the former context (embedded hardware/software implementation) has been addressed extensively in the literature. In this chapter, we focus on the latter context (simulation), which is relatively unexplored in any explicit sense. Our target simulation platforms are single-processor machines including workstations and desktop PCs, which are widely used to host system-level simulation tools. The large-scale and highly multirate nature of todays wireless communication applications is our driving application motivation: for satisfactory simulation, the wireless communication domain requires SDF scheduling techniques that are explicitly and effectively geared towards simulation performance as the primary objective.

The organization of this chapter is as follows: In Section 7.2, we discuss problems that arise from simulating modern wireless communication systems. In Section 7.3, we introduce the *simulation-oriented scheduler* (SOS) for efficient simulation of large-scale, highly multirate synchronous dataflow graphs. We present the overall integration in Section 7.4 and simulation results in Section 7.5.

7.2 Problem Description

Real-world communication and signal processing systems involve complicated physical behaviors, and their behavioral representations may involve hundreds of coarse-grain components that are interconnected in complex topologies, and have heavily multirate characteristics. For example, simulating wireless communication systems involves complex encoder/decoder schemes, modulation/demodulation structures, communication channels, noise, and interference signals. In transmitters, data is converted progressively across representation formats involving bits, symbols, frames, and RF signals. The corresponding conversions are then performed in reverse order at the receiver end. These transmitter-receiver interactions and the data conversions are often highly multirate. In addition, simulating communication channels may involve various bandwidths, noise, and multiple interference signals that may originate from different wireless standards. All of these considerations introduce heavily multirate characteristics across the overall system.

Modeling such communication and signal processing systems usually results in *critical SDF graphs*. By a critical SDF graph, we mean an SDF graph that has: *large scale* (consists of hundreds (or more) of actors and edges); *complex topology* (contains directed and undirected cycles across the graph components); and *heavily multirate behavior* (contains large variations in data transfer rates or component execution rates across graph edges). Here, we define *multirate complexity* as a measure of overall multirate behavior.

Definition 7.1 (Multirate Complexity)**.** Given an SDF graph $G = (V, E)$, its *mul-*

tirate complexity (MC) is defined as an average of its repetitions vector components:

$$MC(G) = \left(\sum_{\forall v \in V} \boldsymbol{q}_G[v] \right) / |V|, \tag{7.1}$$

where $|V|$ is the number of actors in G. In other words, it is an average number of firings per component in one iteration of a minimal periodic schedule.

A complex topology complicates the scheduling process because the properties of data-driven and deadlock-free execution must be ensured. However, large-scale and heavily multirate behavior cause the most serious problems due to the following three related characteristics:

1. **High multirate complexity**. Multirate transitions in an SDF graph, i.e., $\{e \in E \mid prd(e) \neq cns(e)\}$, generally lead to repetition counts that increase exponentially in the number of such transitions [7]. Critical SDF graphs usually have extremely high multirate complexities, even up to the range of millions, as we show in Section 7.5. Such high multirate complexity seriously complicates the scheduling problem (i.e., sequencing large sets of firings for the same actors in addition to sequencing across actors), and has heavy impact on implementation metrics such as memory requirements, schedule length, and algorithm complexity.

2. **Large number of firings**. Highly multirate behavior together with large graph scale generally makes the *number of firings* in a schedule (i.e., the sum of the repetitions vector components) increase exponentially in the number of multirate transitions, and also increase proportionally in the graph size. In

critical SDF graphs, schedules may have millions or even billions of firings, as we show in Section 7.5. As a result, any scheduling algorithm or schedule representation that works at the granularity of individual firings is unacceptable in our context.

3. **Large memory requirements**. Increases in multirate complexity lead to corresponding increases in the overall volume of data transfer and the length of actor firing sequences in an SDF graph. Simulation tools usually run on workstations and PCs where memory resources are abundant. However, due to exponential growth in multirate complexity, algorithms for scheduling and buffer allocation that are not carefully designed may still run out of memory when simulating critical SDF graphs.

In this chapter, we present the *simulation-oriented scheduler* (SOS) for simulating critical SDF graphs in EDA tools. Our objectives include: 1) minimizing simulation run-time; 2) scaling efficiently across various graph sizes, topologies, and multirate complexities; and 3) satisfying memory constraints. Our simulation-oriented scheduler statically computes schedules and buffer sizes with emphasis on low-complexity, static scheduling and memory minimization. Static scheduling and static buffering allow tools to simulate systems and allocate buffers with low set-up cost and low run-time overhead. Low-complexity algorithms scale efficiently across various kinds of SDF graphs and minimize scheduling run-time. In SOS, the memory requirements for storing schedules and buffering data are carefully kept under control to prevent out-of-memory problems, and alleviate virtual memory swapping

behavior, which causes large run-time overhead.

7.3 Simulation-Oriented Scheduler

Our SOS approach integrates several existing and newly-developed algorithms for graph decomposition and scheduling. Figure 7.1 illustrates the overall architecture. Among these techniques, LIAF, APGAN, and DPPO have been developed in [7], and the concept of recursive two-actor graph decomposition has been developed in [45]. These techniques were originally designed for code and data memory minimization in software synthesis, and have not been applied with simulation of critical SDF graphs as an explicit concern. In SOS, we develop a novel integration of these methods, and incorporate into this integrated framework the following new techniques: 1) *cycle-breaking* to achieve fast execution in LIAF [34], 2) *single-rate clustering* (SRC) to alleviate the complexity of APGAN and DPPO, and 3) *buffer-optimal two-actor scheduling* for handling nonzero delays on graph edges in addition to delayless two-actor graphs.

In this section, we present the novel integration as well as the algorithms and theory associated with the new techniques. As discussed in Section 2.1.1, we assume that an SDF schedule is represented in the *looped schedule* format [7].

7.3.1 SDF Clustering

SDF clustering is an important operation in SOS. Given a connected, consistent SDF graph $G = (V, E)$, clustering a connected subset $Z \subseteq V$ into a *supernode*

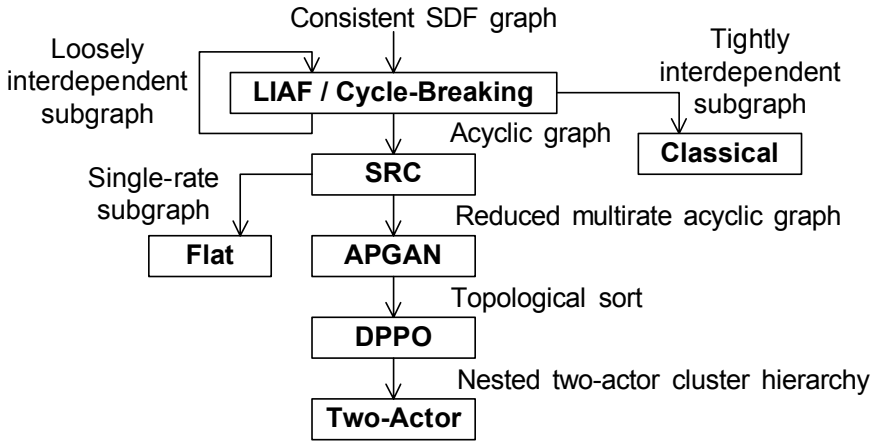


Figure 7.1: Architecture of the simulation-oriented scheduler.

α means: 1) extracting a subgraph $G_\alpha = (Z, \{e \mid src(e) \in Z \text{ and } snk(e) \in Z\})$; and 2) transforming G into a reduced form $G' = (V', E')$, where $V' = V - Z + \{\alpha\}$ and $E' = E - \{e \mid src(e) \in Z \text{ or } snk(e) \in Z\} + E^*$. Here, E^* is a set of "modified" edges in G that originally connect actors in Z to actors outside of Z. More specifically, for every edge e that satisfies ($src(e) \in Z$ and $snk(e) \notin Z$), there is a modified version $e^* \in E^*$ such that $src(e^*) = \alpha$ and $prd(e^*) = prd(e) \times \boldsymbol{q}_{G_\alpha}(src(e))$, and similarly, for every e that satisfies ($src(e) \notin Z$ and $snk(e) \in Z$), there is a modified version $e^* \in E^*$ such that $snk(e^*) = \alpha$ and $cns(e^*) = cns(e) \times \boldsymbol{q}_{G_\alpha}(snk(e))$.

In the transformed graph G', execution of α corresponds to executing one iteration of a minimal periodic schedule for G_α. SDF clustering guides the scheduling process by transforming G into a reduced form G' and isolating a subgraph G_α of G such that G' and G_α can be treated separately, e.g., by using different optimization techniques. SDF clustering [7] guarantees that if we replace every supernode firing α in a schedule $S_{G'}$ for G' with a minimal periodic schedule S_{G_α} for G_α, then the result is a valid schedule for G.

7.3.2 LIAF Scheduling

The *loose interdependence algorithms framework* (LIAF) [7] aims to decompose and break cycles in an SDF graph such that algorithms for scheduling or optimization that are subsequently applied can operated on acyclic graphs. Existence of cycles in the targeted subsystems prevents or greatly restricts application of many useful optimization techniques.

Given a connected, consistent SDF graph $G = (V, E)$, LIAF starts by clustering all *strongly connected components* [1] $Z_1, Z_2, \ldots, Z_N$ into supernodes $\alpha_1, \alpha_2, \ldots, \alpha_N$, and this results in an acyclic graph G_a [16]. For each strongly connected subgraph $G_i = (Z_i, E_i)$, LIAF tries to break cycles by properly removing edges that have "sufficient" delays. An edge $e_i \in E_i$ can be removed in this sense if it has enough initial tokens to satisfy the consumption requirements of its sink actor for a complete iteration of G_i — that is, if $del(e_i) \geq cns(e_i) \times \boldsymbol{q}_{G_i}(snk(e_i))$ — so that scheduling without considering e_i does not deadlock G_i. Such an edge e_i is called an *inter-iteration edge* in our context.

Now suppose that ${G_i}^*$ denotes the graph that results from removing all inter-iteration edges from the strongly connected subgraph G_i. G_i is said to be *loosely interdependent* if ${G_i}^*$ is not strongly connected, and G_i is said to be *tightly interdependent* if G_i^* is strongly connected. If G_i is found to be loosely interdependent, then LIAF is applied recursively to the modified version G_i^* of G_i.

[1] A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $Z \subseteq V$ such that for every pair of vertices u and v in Z, there is a path from u to v and a path from v to u.

In our application of LIAF in SOS, tightly interdependent subgraphs are scheduled by classical SDF scheduling, which is discussed in more detail in Section 7.3.4, and the acyclic graphs that emerge from the LIAF decomposition process are further processed by the techniques developed in Section 7.3.5 through 7.3.9. The process that we employ for breaking cycles in strongly components, which has been described intuitively above, is described in more detail in Section 7.3.3 below.

7.3.3 Cycle-Breaking

Careful decomposition of strongly connected SDF graphs into hierarchies of acyclic graphs — a process that is referred to as *subindependence partitioning* or *cycle-breaking* — is a central part of the LIAF framework. LIAF does not specify the exact algorithm that is used to break cycles, but rather specifies the constraints that such an algorithm must satisfy so that schedulers derived from the framework can construct single appearance schedules whenever they exist and satisfy other useful properties [7, 5].

For using LIAF in SOS, we have developed the cycle-breaking algorithm [34], which is presented in Figure 7.2. Particularly, our cycle-breaking algorithm is designed to be well suited for the acyclic scheduling techniques in Section 7.3.5 through 7.3.9, and it is also designed for low complexty, which is important for use in SOS, as well as in other environments where scheduling runtime is critical.

In Figure 7.2, given a strongly connected SDF graph $G = (V, E)$ to CYCLE-BREAKING, we first remove all inter-iteration edges from G (lines 2-4). If G is

```
CYCLE-BREAKING(G ≡ (V, E)) /*The input G is a strongly connected SDF graph*/
 1  E' ← ∅
 2  for e ∈ E
 3     if del(e) ≥ cns(e) × q_G[snk(e)]     E ← E − e,  E' ← E' + e     end
 4  end
 5  if IS-CONNECTED(G)
 6     {SCC_1, SCC_2, ..., SCC_N} ← TOPOLOGICALLY-SORTED-SCC(G)
 7     if N = 1     G is tightly interdependent, E ← E + E', ...
 8     else
 9        for e ∈ E'
10           if !(src(e) ∉ SCC_1 and snk(e) ∈ SCC_1)     E ← E + e,  E' ← E' − e     end
11        end
12        G is no longer strongly connected ...
13     end
14  else
15     {CC_1, CC_2, ..., CC_M} ← CONNECTED-COMPONENTS(G)
16     {SCC_1, SCC_2, ..., SCC_P} ← TOPOLOGICALLY-SORTED-SCC(G_CC_1 ≡ (CC_1, E_CC_1))
17     for e ∈ E'
18        if !(src(e) ∉ SCC_1 and snk(e) ∈ SCC_1)     E ← E + e,  E' ← E' − e     end
19     end
20     G is no longer strongly connected ...
21  end
```

Figure 7.2: Cycle-breaking algorithm.

connected (line 5), we compute the strongly connected components SCC_1, SCC_2, ..., SCC_N of G in topologically sorted order (line 6). By a topologically sorted order of SCCs, we mean a topological sort [2] of the acyclic graph that results from clustering the SCCs in G. In addition, for a vertex that does not belong to any SCC that contains at least two vertices, we say that this vertex is an SCC by itself.

If G is still strongly connected ($N = 1$ in line 7), we conclude that G is tightly interdependent; restore G to its original state (before any edge removals); and mark it for processing by the tightly interdependent scheduling techniques, e.g., classical scheduling discussed in Chapter 3. On the other hand, if G is connected, but not strongly connected ($N > 1$ in line 7), then we put all previously removed edges

[2] A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of V such that for every edge (u, v) in G, u appears before v in the ordering.

(which are stored in E') back in G, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 9-11).

If G becomes disconnected after removing all inter-iteration edges (that is, if control passes to the *else* branch rooted at line 14), then we compute the connected components (CCs) $CC_1, CC_2, \ldots, CC_M$ (line 15). Here, $M > 1$, and the CCs can be ordered arbitrarily. Next, we compute the strongly connected components $SCC_1, SCC_2, \ldots, SCC_P$ $(P \geq 1)$ in some topologically sorted order for one of the connected subgraphs $G_{CC_1} = \{CC_1, E_{CC_1}\}$ (line 16). Lastly, we return all previously-removed edges back to G, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 17-19), and complete the process.

The following theorem proves the correctness of the cycle-breaking algorithm.

Theorem 7.2. *Suppose a strongly connected SDF graph $G = (V, E)$ is applied as input to the CYCLE-BREAKING algorithm, then G is determined to be tightly interdependent in line 7, is determined (after modification) to not be strongly connected in line 12, or is determined (again, after modification) to not be strongly connected in line 20.*

Proof. CASE I: In line 7, G is tightly interdependent because after removing all inter-iteration edges (lines 2-4), it is still strongly connected (line 6-7).

CASE II: Just after line 8, the modified version of G (after removing all inter-iteration edges) is connected and has $N > 1$ SCCs, $SCC_1, SCC_2, \ldots, SCC_N$, ordered in a topologically sorted fashion. We can then determine that G is not strongly connected (since there are $N > 1$ SCCs), and there is no edge from $\{V - SCC_1\}$

to SCC_1 (since SCC_1 is the first SCC in topologically sorted order). By putting back all previously removed edges, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 9-11), the resulting graph G (line 12) is not strongly connected because for any $u \in SCC_1$ and $v \in \{V - SCC_1\}$, there is a path from u to v (since the original input G is strongly connected), but no path from v to u (since there is no edge from $\{V - SCC_1\}$ to SCC_1).

CASE III: Just after line 16, the modified version of G (after removing all inter-iteration edges) is disconnected, and SCC_1 here is the first SCC in topologically sorted order in the connected subgraph $G_{CC_1} \equiv \{CC_1, E_{CC_1}\}$. We can then derive that there is no edge from $\{V - SCC_1\}$ to SCC_1 (since there is no edge between CCs, and SCC_1 is the first SCC in topologically sorted order in G_{CC_1}). By putting back all previously removed edges, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 17-19), the resulting graph G (line 20) is connected but not strongly connected because for any $u \in SCC_1$ and $v \in \{V - SCC_1\}$, there is a path from u to v (since the original input G is strongly connected), but no path from v to u (since there is no edge from $\{V - SCC_1\}$ to SCC_1). □

The following theorem establishes key properties provided by our CYCLE-BREAKING algorithm.

Theorem 7.3. *If a loosely interdependent, strongly connected SDF graph $G = (V, E)$ is applied as input to the CYCLE-BREAKING algorithm, then the resulting graph G is connected. Also, suppose that $SCC'_1, SCC'_2, \ldots, SCC'_L$ are the $L > 1$ SCCs in* ***any*** *topologically sorted order of the resulting graph G (line 12 or line 20). Then the*

edges removed by the CYCLE-BREAKING algorithm are edges from $\{V - SCC'_1\}$ *to* SCC'_1*. Furthermore,* SCC'_1 *is equal to* SCC_1 *in line 6 or line 16.*

Proof. Continuing from the proof of Theorem 7.2 for both CASE II and CASE III, we can derive that 1) SCC_1 is a strongly connected component in the resulting graph G; and for any $u \in SCC_1$ and $v \in \{V - SCC_1\}$, 2) there is a path from u to v, but 3) there is no path from v to u. As a result, the resulting graph G is connected. In addition, SCC_1 must be the first SCC in *any* topologically sorted order of the resulting graph G, i.e., $SCC_1 = SCC'_1$; and the removed edges, i.e., inter-iteration edges from $\{V - SCC_1\}$ to SCC_1, must be edges from succeeding SCCs, $SCC'_2, SCC'_3, \ldots, SCC'_L$, to the first SCC'_1 in the resulting graph G. □

The following theorem pertains to the complexity of our cycle-breaking algorithm.

Theorem 7.4. *Given a strongly connected SDF graph* $G = (V, E)$*, the complexity of the CYCLE-BREAKING algorithm is* $\Theta(|V| + |E|)$*.*

Proof. Determining whether a graph is connected (IS-CONNECTED) as well as computing connected components of a disconnected graph (CONNECTED-COMPONENTS) can be implemented in linear time (i.e., in time that is linear in the number of actors and edges in G). This can be done, for example, by using depth-first search. A linear time algorithm to compute SCCs of a directed graph in topologically sorted order (TOPOLOGICALLY-SORTED-SCC) can be found in [16]. Computing the repetitions vector of an SDF graph can also be implemented in linear time [7]. Furthermore, with efficient data structures, operations in lines

2-4, lines 9-11, and lines 17-19, can be implemented in linear time. As a result, the complexity of CYCLE-BREAKING is $\Theta(|V| + |E|)$. □

With the CYCLE-BREAKING algorithm, operations for decomposing and breaking cycles in LIAF can be implemented in time that is linear in the number of actors and edges in the input SDF graph.

7.3.4 Classical SDF Scheduling

As described in Chapter 3, *classical SDF scheduling* is a demand-driven, minimum-buffer scheduling heuristic. By simulating demand-driven dataflow behavior (i.e., by deferring execution of an actor until output data from it is needed by other actors), we can compute a buffer-efficient actor firing sequence and the associated buffer sizes. The complexity of classical SDF scheduling is not polynomially-bounded in the size of the input graph, and we use it only as a backup process for scheduling tightly interdependent subgraphs from LIAF. Fortunately, this does not cause any major limitation in SOS because tightly interdependent subgraphs arise very rarely in practice [7]. For example, we have tested SOS on a suite of 126 wireless network designs and 267 wireless communication designs, and among all of these designs, no tightly interdependent subgraphs were found.

7.3.5 Single-Rate Clustering

Intuitively, a single-rate subsystem in an SDF graph is a subsystem in which all actors execute at the same average rate. In practical communication and signal

processing systems, single-rate subsystems arise commonly, even within designs that are heavily multirate at a global level. In precise terms, an SDF graph is a single-rate graph if for every edge e, we have $prd(e) = cns(e)$. Since clustering single-rate subsystems does not increase production and consumption rates at the interface of the resulting supernodes, we have developed the *single-rate clustering* (SRC) technique to further decompose an acyclic graph into a reduced (smaller) multi-rate version along with several single-rate subgraphs. Due to their simple structure, single-rate subgraphs can be scheduled and optimized effectively by the accompanying *flat scheduling* (Section 7.3.6) algorithm in a very fast manner. Furthermore, the reduced multirate graph, which is scheduled using the more intensive techniques described in Sections 7.3.7 through 7.3.9, takes less time to schedule due to its significantly smaller size — that is, since each single-rate subsystem is abstracted as a single actor (supernode).

Definition 7.5 (Single-Rate Clustering)**.** Given a connected, consistent, acyclic SDF graph $G = (V, E)$, the *single-rate clustering* (SRC) technique clusters disjoint subsets $R_1, R_2, \ldots, R_N \subseteq V$ such that: 1) in the subgraph $G_i = (R_i, E_i)$, we have that $\forall e_i \in E_i = \{e \mid src(e) \in R_i \text{ and } snk(e) \in R_i\}$, $prd(e_i) = cns(e_i)$; 2) the clustering of R_i does not introduce any cycles into the clustered version of G; 3) R_i satisfies $|R_i| > 1$ (i.e., R_i contains at least two actors); and 4) each R_i contains a maximal set of actors that satisfy all of the three conditions above. Such R_is are defined as *single-rate subsets*; and such G_is are defined as *single-rate subgraphs.*

The targeting of "maximal" clusters in the fourth condition is important in

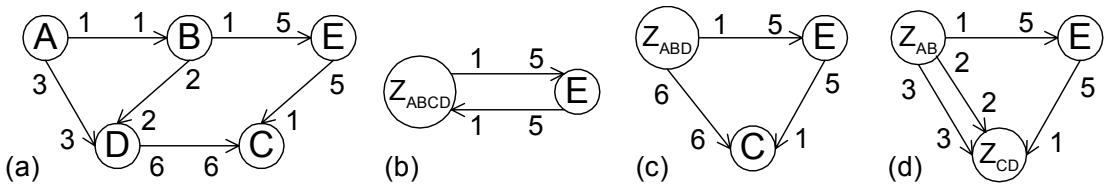


Figure 7.3: Single-rate clustering examples.

order to effectively reduce the size of the clustered graph, and maximize the extent of the overall system that can be handled with the streamlined techniques available for single-rate graphs.

Simply clustering a set of actors that is connected through single-rate edges may introduce cycles in the clustered graph. Figure 7.3 illustrates how this simple strategy fails. Nodes A, B, C, and D in Figure 7.3.(a) are connected by single-rate edges, and clustering them will result in a cyclic graph as shown in Figure 7.3.(b). In contrast, Figure 7.3.(c) and Figure 7.3.(d) present two acyclic SDF graphs after valid single-rate clustering. The following theorem provides a precise condition for the introduction of a cycle by a clustering operation.

Theorem 7.6 (Cycle-Free Clustering Theorem)**.** *Given a connected, acyclic SDF graph $G = (V, E)$, clustering a connected subset $R \subseteq V$ introduces a cycle in the clustered version of G if and only if there is a path $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ ($n \geq 3$) in G such that $v_1 \in R$, $v_n \in R$, and $v_2, \ldots, v_{n-1} \in \{V - R\}$. Clustering R is cycle-free if and only if no such a path exists.*

Proof. Without loss of generality, suppose that we cluster R into a supernode α, and this results in a subgraph G_α and the clustered version G'. Based on SDF clustering, as discussed in Section 7.3.1, we have: 1) for every edge

$e \in \{e \mid src(e) \in R \text{ and } snk(e) \notin R\}$, it becomes an output edge $e' = (\alpha, snk(e))$ of α in G', and every output edge of α comes from this transformation; and 2) for every edge $e \in \{e \mid src(e) \notin R \text{ and } snk(e) \in R\}$, it becomes an input edge $e' = (src(e), \alpha)$ of α in G', and every input edge of α comes from this transformation. Therefore, a path $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ in G, where $v_1 \in R$, $v_n \in R$, and $v_2 \cdots \in \{V - R\}$, becomes a cycle $\alpha \rightarrow v_2 \rightarrow \cdots \rightarrow \alpha$ in G'. In addition, a cycle containing α in G' can only come from such a path in G. □

We have developed the SRC algorithm as presented in Figure 7.4. Given a connected, acyclic SDF graph $G = (V, E)$, we first duplicate G into $G' = (V', E')$ to prevent us from modifying G before actually clustering the single-rate subsets. Given G' and an actor v, the subroutine SRS (single-rate subset) returns a single-rate subset that contains v or returns $\varnothing$ if no such single-rate subset exists for v. In lines 2-7, all single-rate subsets $R_1, R_2, \ldots, R_N$ are computed, and particularly, the "next actor" in line 3 refers to the next actor that has not yet been visited in the *remaining* V' after each call of SRS. In line 8, we cluster $R_1, R_2, \ldots, R_N$ in G by repeatedly calling the SDF clustering operation CLUSTER.

SRS iteratively determines whether an adjacent actor x of v belongs to the single-rate subset R in a breadth-first fashion. An adjacent, non-clustered successor $x = snk(a)$ of v in line 11 can be included in R if 1) every edge connecting v to x is single-rate (i.e., $B = \varnothing$ in line 15) and 2) clustering v and x does not introduce a cycle (i.e., IS-CYCLE-FREE(G',v,x) returns TRUE). If both criteria are satisfied, x is included in R (line 18), and G' is transformed to mimic the effect of clustering

```
SRC(G ≡ (V, E)) /*The input G is a connected acyclic SDF graph*/
 1   G' ≡ (V', E') ← G
 2   i ← 1
 3   for the next actor v ∈ V'
 4       R_i ← SRS(G', v)
 5       if R_i ≠ ∅     i ← i + 1     end
 6   end
 7   N ← i − 1
 8   for i from 1 to N     CLUSTER(G, R_i)     end

SRS(G', v)
 9   R ← {v}
10   for the next edge a ∈ {in(v) + out(v)}
11       if src(a) = v and x ← snk(a) is not in any subset
12           A ← {e ∈ in(x) | src(e) = v and prd(e) = cns(e)}
13           B ← {e ∈ in(x) | src(e) = v and prd(e) ≠ cns(e)}
14           C ← {e ∈ in(x) | src(e) ≠ v}
15           if B = ∅ and IS-CYCLE-FREE(G', v, x)
16               for each e ∈ out(x)     src(e) ← v     end
17               for each e ∈ C     snk(e) ← v     end
18               R ← R + {x}, E' ← E' − A, V' ← V' − {x}
19           end
20       else if snk(a) = v and x ← src(a) is not in any subset
21           A ← {e ∈ out(x) | snk(e) = v and prd(e) = cns(e)}
22           B ← {e ∈ out(x) | snk(e) = v and prd(e) ≠ cns(e)}
23           C ← {e ∈ out(x) | snk(e) ≠ v}
24           if B = ∅ and IS-CYCLE-FREE(G', x, v)
25               for each e ∈ in(x)     snk(e) ← v     end
26               for each e ∈ C     src(e) ← v     end
27               R ← R + {x}, E' ← E' − A, V' ← V' − {x}
28           end
29       end
30   end
31   if R = {v}     return ∅     else     return R     end

IS-CYCLE-FREE(G', y, z)
32   E_yz ← {e | src(e) = y and snk(e) = z}
33   if {out(y) − E_yz} = ∅ or {in(z) − E_yz} = ∅       return TRUE
34   else     return ! IS-REACHABLE((V', {E' − E_yz}), y, z)
35   end
```

Figure 7.4: Single-rate clustering (SRC) algorithm.

x in lines 16-18. After that, the actor v in G' represents the R-clustered supernode. On the other hand, for an adjacent predecessor $x = src(a)$ of v, similar operations are performed in lines 20-29. Note that after each iteration, v's incident edges $\{in(v) + out(v)\}$ in line 10 may have been changed because the on-line topology transformation removes and inserts incident edges of v, and particularly, the "next" edge in line 10 refers to the next edge that has not yet been visited.

IS-CYCLE-FREE determines whether clustering a source node y and a sink node z of an edge in G' is cycle-free based on Theorem 7.6 (i.e., by checking whether there is a path from y to z through other actors). If all output edges of y connect to z or all input edges of z connect from y, we can immediately determine that no such path exists (line 33). Otherwise in line 34, we test to ensure that z is *not* reachable from y when all edges connecting y to z are removed.

Property 7.7. *The set R returned by SRS in the SRC algorithm is a single-rate subset.*

Proof. The set R is a single-rate subset if it satisfies the conditions in Definition 7.5. Because an adjacent actor x of v (in line 11 and line 20) can be included in R if every edge connected between x and v is single-rate and clustering x and v is cycle-free, and since v represents the up-to-date R-cluster, condition 1 and 2 in Definition 7.5 are satisfied. Condition 3 is simply checked by line 31. Condition 4 can be satisfied if at the end of iterations, every surrounding edge of R has been searched for determining whether the adjacent actor x belongs to the single-rate subset. This is true because SRS iterates over every incident edge of v in a breadth-first way and

updates v's incident edges in each iteration. □

Before discussing the complexity of the SRC algorithm, and the complexity of the algorithms in the following sections, we make the assumption that every actor has a constant (limited) number of input and output edges, i.e., $|V|$ and $|E|$ are within a similar range. This is a reasonable assumption because actors in simulation tools are usually pre-defined, and practical SDF graphs in communications and signal processing domains are sparse in their topology [7].

Property 7.8. *The complexity of the SRC algorithm is* $O(|E|^2)$*, where* E *denotes the set of edges in the input graph.*

Proof. By the combination of the for loop in line 3 and the for loop in line 10, an edge can be visited by line 10 once (if clustered), twice (if not clustered), or none (if there are parallel edges between two nodes). Therefore, the total number of edges examined in line 10 is $O(|E|)$. With efficient data structures and the assumption that every actor has a limited number of incident edges, operations (lines 11-29) within the for loop in line 10 require constant time, except for IS-CYCLE-FREE, which takes $O(|E'| + |V'|)$ time. As a result, the running time to compute all single-rate subsets (lines 3-6) is $O(|E|^2)$. In the last step, the complexity to cluster $R_1, R_2, \ldots, R_N$ in line 8 is bounded by $O(|V| + |E|)$. Therefore, the complexity of the SRC algorithm is $O(|E|^2)$. □

7.3.6 Flat Scheduling

Given a consistent, acyclic SDF graph $G = (V, E)$, a valid single appearance schedule S can be easily derived by *flat scheduling*. Flat scheduling simply computes a topological sort $v_1\ v_2 \cdots v_{|V|}$ of G, and iterates each actor v_i $\boldsymbol{q}_G[v_i]$ times in succession. More precisely, the looped schedule constructed from the topological sort in flat scheduling is $S = (\boldsymbol{q}_G[v_1]\ v_1)\ (\boldsymbol{q}_G[v_2]\ v_2) \cdots (\boldsymbol{q}_G[v_{|V|}]\ v_{|V|})$. The complexity of flat scheduling is $\mathrm{O}(|V| + |E|)$ — topological sort can be performed in linear time [16], and the repetitions vector can also be computed in linear time [7]. However, in general, the memory requirements of flat schedules can become very large in multirate systems [7].

We apply flat scheduling only to single-rate subgraphs, which do not suffer from the memory penalties that are often associated with flat scheduling in general SDF graphs. This is because in a single-rate subgraph, each actor only fires once within a minimal periodic schedule. Thus, the buffer size of each single rate edge e can be set to $\mathit{buf}(e) = \mathit{prd}(e) + \mathit{del}(e)$, which is the minimum achievable size whenever $\mathit{del}(e) < \mathit{prd}(e)$.

7.3.7 APGAN Scheduling

In general, computing buffer-optimal topological sorts in SDF graphs is NP-hard [61]. The *acyclic pairwise grouping of adjacent nodes* (APGAN) [7] technique is an adaptable (to various cost functions) heuristic to generate topological sorts. Given a consistent, acyclic SDF graph $G = (V, E)$, APGAN iteratively selects and

clusters adjacent pairs of actors until the top-level clustered graph consists of a single supernode. The clustering process is guided by a cost function $f(e)$ that estimates the impact of clustering of actors $src(e)$ and $snk(e)$ into a supernode. In each iteration, APGAN clusters an adjacent pair $\{src(e), snk(e)\}$ in the current version of G such that 1) clustering this pair does not introduce cycles in the clustered graph; and 2) the applied cost function f is maximized for e over all edges e^* for which $\{src(e^*), snk(e^*)\}$ can be clustered without introducing cycles. Once the clustering process is complete, a topological sort is obtained through depth-first, source-to-sink traversal of the resulting cluster hierarchy.

In our incorporation of APGAN in SOS, we use the following as the cost function f: $\gcd(\boldsymbol{q}_G[src(e)], \boldsymbol{q}_G[snk(e)])$, where gcd represents the *greatest common divisor* operator. This cost function has been found to direct APGAN towards solutions that are efficient in terms of buffering requirements [7].

The complexity of APGAN is $O(|V|^2|E|)$ [7]. At first, this appears relatively high in relation to the objective of low complexity in this work. However, due to the design of our SOS framework, which applies the LIAF and SRC decomposition techniques, $|V|$ and $|E|$ are typically much smaller in the instances of APGAN that result during operation of SOS compared to the numbers of actors and edges in the overall SDF graph.

7.3.8 DPPO Scheduling

Given a topological sort $L = v_1\ v_2 \cdots v_{|V|}$ of an acyclic SDF graph $G = (V, E)$, the *dynamic programming post optimization* (DPPO) technique [7] constructs a memory-efficient hierarchy of nested two-actor clusters. This hierarchy is constructed in a bottom-up fashion by starting with each two-actor subsequence $v_i\ v_{i+1}$ in the topological sort and progressively optimizing the decomposition of longer subsequences. Each l-actor subsequence $L_{i,j} = v_i\ v_{i+1} \cdots v_{j=i+l-1}$ is "split" into two shorter "left" ($L_{i,k} = v_i\ v_{i+1} \cdots v_k$) and "right" ($L_{k+1,j} = v_{k+1}\ v_{k+2} \cdots v_j$) subsequences. In particular, the split position k is chosen to minimize the buffer requirement of $L_{i,j}$ — i.e., the sum of buffer requirements associated with the left and right subsequences plus the buffer requirements for the set $E_{i,j,k} = \{e \mid src(e) \in \{v_i, v_{i+1}, \ldots, v_k\} \text{ and } snk(e) \in \{v_{k+1}, v_{k+2}, \ldots, v_j\}\}$ of edges that cross from left to right. Through dynamic programming, where the outer loop l iterates from 2 to $|V|$, the middle loop i iterates from 1 to $|V| - l + 1$, and inner loop k iterates from i to $i + l - 2$, the best split position and minimal buffer requirement for every subsequence can be derived. A memory-efficient hierarchy is then built in a top-down fashion by starting from the topological sort $L_{1,|V|}$, and recursively clustering the left $L_{i,k}$ and right $L_{k+1,j}$ subsequences through the best split position k of $L_{i,j}$.

In SOS, we have developed an adapted DPPO where each split k of $L_{i,j}$ is interpreted as a two-actor SDF graph $G_{i,j,k} = (\{\alpha_{i,k}, \alpha_{k+1,j}\}, E'_{i,j,k})$, where the left $L_{i,k}$ and right $L_{k+1,j}$ subsequences make up the two hierarchical actors $\alpha_{i,k}$ and $\alpha_{k+1,j}$,

and every edge $e' \in E'_{i,j,k}$ is a transformation from the corresponding edge $e \in E_{i,j,k}$ such that $prd(e') = prd(e) \times \boldsymbol{q}_G[src(e)]/\text{gcd}(\boldsymbol{q}_G[v_i], \boldsymbol{q}_G[v_{i+1}], \ldots, \boldsymbol{q}_G[v_k])$ and $cns(e') = cns(e) \times \boldsymbol{q}_G[snk(e)]/\text{gcd}(\boldsymbol{q}_G[v_{k+1}], \boldsymbol{q}_G[v_{k+2}], \ldots, \boldsymbol{q}_G[v_j])$ based on SDF clustering concepts. This two-actor graph is further optimized, after DPPO, by the *buffer-optimal two-actor scheduling* algorithm discussed in Section 7.3.9. Furthermore, the optimal buffer requirements for $E'_{i,j,k}$ can be computed in constant time (based on Theorem 7.17, which is developed below) without actually computing a two-actor schedule for each split.

DPPO can be performed in $O(|V|^3)$ time [7]. The complexity of our adapted DPPO is also $O(|V|^3)$ because the cost function — that is, the optimal buffer requirement of each two-actor graph $G_{i,j,k}$ — can be computed in constant time. Also, as with the techniques discussed in the previous section, the value of $|V|$ for instances of our adapted DPPO technique is relatively small due to the preceding stages of LIAF- and SRC-based decomposition.

7.3.9 Buffer-Optimal Two-Actor Scheduling

The concept of recursive two-actor scheduling for a nested two-actor SDF hierarchy was originally explored in [45]. For *delayless* SDF graphs, the resulting schedules are proven to be buffer-optimal at each (two-actor) level of the cluster hierarchy. These schedules are also polynomially bounded in the graph size. However, the algorithm in [45] does not optimally handle the scheduling flexibility provided by edge delays, and therefore, it does not always achieve minimum buffer sizes in

presence of delays. We have developed a new *buffer-optimal two-actor scheduling* algorithm that computes a buffer-optimal schedule for a *general* (with or without delays), consistent, acyclic, two-actor SDF graph. This algorithm is applied in SOS to schedule each two-actor subgraph in the DPPO hierarchy. An overall schedule is then constructed by recursively traversing the hierarchy and replacing every supernode firing by the corresponding two-actor sub-schedule. In this subsection, we present definitions, analysis, and an overall algorithm that are associated with our generalized, two-actor scheduling approach.

Property 7.9. *A consistent, acyclic, two-actor SDF graph* $G = (\{v_{src}, v_{snk}\}, E)$ *has a general form as shown in Figure 7.5, where for each* $e_i \in E$, $src(e_i) = v_{src}$, $snk(e_i) = v_{snk}$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, *and* $c^* = c_i/g_i$. *For consistency, the coprime positive integers* p^* *and* c^* *must satisfy* $p_i/c_i = p^*/c^*$ *for every* $e_i \in E$.

Definition 7.10 (Primitive Two-Actor SDF Graph)**.** Given a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ as described in Property 7.9, its *primitive form* is defined as a two-actor, single-edge SDF graph $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$ as shown in Figure 7.6, where $src(e^*) = v_{src}$, $snk(e^*) = v_{snk}$, $prd(e^*) = p^*$, $cns(e^*) = c^*$, $\gcd(p^*, c^*) = 1$, and $del(e^*) = d^* = \min_{e_i \in E}(\lfloor d_i/g_i \rfloor)$. The values p^*, c^*, and d^* are defined as the *primitive production rate*, *primitive consumption rate*, and *primitive delay* of G, respectively. An edge e_i that satisfies $\lfloor d_i/g_i \rfloor = d^*$ is called a *maximally-constrained edge* of G.

Here, we also define some notations that are important to our development of

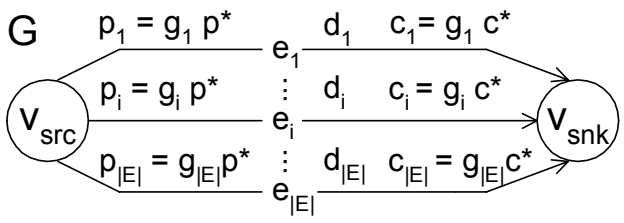


Figure 7.5: Consistent, acyclic, two-actor SDF graph.

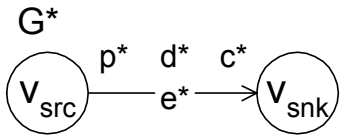


Figure 7.6: Primitive two-actor SDF graph.

two-actor scheduling. Suppose that we are given a consistent SDF graph $G = (V, E)$ and a valid minimal periodic schedule S for G. By a *firing index* for S, we mean a non-negative integer that is less than or equal to the sum Q_G of repetitions vector components for G (i.e., $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$). In the context of S, a firing index value of k represents the kth actor execution within a given iteration (minimal period) of the execution pattern derived from repeated executions of S.

Now let $\sigma(S, k)$ denote the actor associated with firing index k for the schedule S; let $\tau(S, v, k)$ denote the firing count of actor v up to firing index k (i.e., the number of times that v is executed in a given schedule iteration up to the point in the firing sequence corresponding to k); and let

$$tok_G(S, e, k) = \tau(S, src(e), k) \times prd(e) - \tau(S, snk(e), k) \times cns(e) + del(e) \quad (7.2)$$

denote the number of tokens queued on edge $e \in E$ immediately after the actor firing associated with firing index k in any given schedule iteration. Firing index

0 represents the initial state: for $k = 0$, $\sigma(S, 0)$ is defined to be $\varnothing$ (the "null actor"), $\tau(S, v, 0)$ is defined to be 0, and $tok_G(S, e, 0)$ is defined as $del(e)$. Note that from the properties of periodic schedules, the values of σ, τ, and tok_G are uniquely determined by k, and are not dependent on the schedule iteration [7]. The repeated execution of S leads to an infinite sequence $x_1, x_2, \ldots$ of actor executions, where each x_i corresponds to firing index $((i-1) \bmod Q_G) + 1$.

For example, suppose that we have an SDF graph $G = (\{a, b\}, \{e = (a, b)\})$, where $prd(e) = 7$, $cns(e) = 5$, and $del(e) = 0$. Suppose also that we have the schedule $S = (1(2ab)(1a(2b)))(1(1ab)(1a(2b)))$. Then we can unroll S into a firing sequence $ababbabbabb$, where $\sigma(S, 1) = a$, $\sigma(S, 6) = b$, $\tau(S, a, 6) = 3$, $\tau(S, b, 6) = 3$, $tok_G(S, e, 0) = 0$, and $tok_G(S, e, 2) = 2$.

The following lemma is useful in simplifying scheduling and analysis for acyclic, two-actor SDF graphs.

Lemma 7.11. *A schedule S is a valid minimal periodic schedule for a consistent, acyclic, two-actor SDF graph G if and only if S is a valid minimal periodic schedule for the primitive form G^* of G.*

Proof. Without loss of generality, suppose G is in a general form as shown in Figure 7.5, and suppose Figure 7.6 represents its primitive form G^*. First, we prove the *only if* direction. S is a valid minimal periodic schedule for G if and only if 1) S fires v_{src} $\boldsymbol{q}_G[v_{src}]$ times and fires v_{snk} $\boldsymbol{q}_G[v_{snk}]$ times, and 2) S is deadlock-free. Because $\boldsymbol{q}_G[v_{src}] = c^* = \boldsymbol{q}_{G^*}[v_{src}]$ and $\boldsymbol{q}_G[v_{snk}] = p^* = \boldsymbol{q}_{G^*}[v_{snk}]$, we know that S fires v_{src} $\boldsymbol{q}_{G^*}[v_{src}]$ times and fires v_{snk} $\boldsymbol{q}_{G^*}[v_{snk}]$ times in G^* — (A). Furthermore, S is

deadlock-free for G if and only if the number of tokens queued on every edge e_i is greater than or equal to c_i before every firing of v_{snk}. In other words, for every $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, $tok_G(S, e_i, k_{snk} - 1) \geq c_i$ is true for every e_i. Through Equation (7.3), where $\tau(S, v_{src}, k_{snk} - 1)$ is denoted as a, and $\tau(S, v_{snk}, k_{snk} - 1)$ is denoted as b, we can derive that $tok_{G^*}(S, e^*, k_{snk} - 1) \geq c^*$ for every k_{snk} — that is, the number of tokens queued on e^* before every firing of v_{snk} is greater than or equal to c^* in G^*, so S is deadlock-free for G^* — (B). Based on (A) and (B), the *only if* direction is proved.

$$\begin{aligned}
&\forall i \quad tok_G(S, e_i, k_{snk}-1) = p_i \times a - c_i \times b + d_i \geq c_i \\
\Leftrightarrow\ &\forall i \quad p^* \times g_i \times a - c^* \times g_i \times b + \lfloor d_i/g_i \rfloor \times g_i + d_i \bmod g_i \geq c^* \times g_i \\
\Leftrightarrow\ &\forall i \quad p^* \times a - c^* \times b + \lfloor d_i/g_i \rfloor \geq c^* \\
\Leftrightarrow\ &p^* \times a - c^* \times b + d^* = tok_{G^*}(S, e^*, k_{snk}-1) \geq c^*
\end{aligned} \tag{7.3}$$

The *if* direction can be proved in a similar manner by applying the same derivations in reverse order and based on the reverse direction in Equation (7.3). We omit the details for brevity. □

Definition 7.12 (SASAP Schedule)**.** A sink-as-soon-as-possible (SASAP) schedule S for a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ is defined as a valid minimal periodic schedule such that: 1) S fires v_{src} $\boldsymbol{q}_G[v_{src}]$ times and fires v_{snk} $\boldsymbol{q}_G[v_{snk}]$ times; 2) for every firing index $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, we have that $tok_G(S, e_i, k_{snk} - 1) \geq c_i$ for every $e_i \in E$, and $\tau(S, v_{snk}, k_{snk} - 1) < \boldsymbol{q}_G[v_{snk}]$; and 3) for every firing index $k_{src} \in \{k \mid \sigma(S,k) = v_{src}\}$, either there exists an edge $e_i \in E$ such that $tok_G(S, e_i, k_{src} - 1) < c_i$ or $\tau(S, v_{snk}, k_{src} - 1) = \boldsymbol{q}_G[v_{snk}]$. If an

actor firing subsequence (sub-schedule) S' satisfies 2) and 3), we say that S' fires v_{snk} as-soon-as-possible (ASAP).

Intuitively, an SASAP schedule can be viewed as a specific form of demand-driven schedule for periodic scheduling of acyclic, two actor SDF graphs. An SASAP schedule defers execution of the source actor in an acyclic, two-actor configuration until the sink actor does not have enough input data to execute. This form of scheduling leads to minimum buffer schedules as we state in the following property because tokens are produced by the source actor only when necessary.

Property 7.13. *An SASAP schedule for a consistent, acyclic, two-actor SDF graph G is a minimum buffer schedule for G.*

The following lemma relates SASAP schedules and primitive forms.

Lemma 7.14. *A schedule S is an SASAP schedule for a consistent, acyclic, two-actor SDF graph G if and only if S is an SASAP schedule for the primitive form G^* of G.*

Proof. The validity and minimal periodic property of S in both directions is proved in Lemma 7.11. Here, we prove the SASAP property. Again, without loss of generality, suppose G is in a general form as shown in Figure 7.5, and suppose Figure 7.6 represents its primitive form G^*. We first prove the *only if* direction. S is an SASAP schedule for G if and only if 1) for every $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, $tok_G(S, e_i, k_{snk} - 1) \geq c_i$ for every e_i and $\tau(S, v_{snk}, k_{snk} - 1) < \boldsymbol{q}_G[v_{snk}]$, and 2) for every $k_{src} \in \{k \mid \sigma(S,k) = v_{src}\}$, either there exists at least one e_i where

$tok_G(S, e_i, k_{src} - 1) < c_i$ or $\tau(S, v_{snk}, k_{src} - 1) = \boldsymbol{q}_G[v_{snk}]$. Through Equation (7.3) and based on $\boldsymbol{q}_G[v_{snk}] = p^* = \boldsymbol{q}_{G^*}[v_{snk}]$, we can derive that for every $k_{snk} \in \{k \mid \sigma(S, k) = v_{snk}\}$, $tok_{G^*}(S, e^*, k_{snk} - 1) \geq c^*$ and $\tau(S, v_{snk}, k_{snk} - 1) < \boldsymbol{q}_{G^*}[v_{snk}]$. Furthermore, through Equation (7.4), where $\tau(S, v_{src}, k_{src} - 1)$ is denoted as a, and $\tau(S, v_{snk}, k_{src} - 1)$ is denoted as b, we can derive that for every $k_{src} \in \{k \mid \sigma(S, k) = v_{src}\}$, either $tok_{G^*}(S, e^*, k_{src} - 1) < c^*$ or $\tau(S, v_{snk}, k_{src} - 1) = \boldsymbol{q}_{G^*}[v_{snk}]$. Therefore, if S is an SASAP schedule for G, then S is an SASAP schedule for G^*.

$$
\begin{aligned}
& \exists i \;\; tok_G(S, e_i, k_{src}-1) = p_i \times a - c_i \times b + d_i < c_i \\
\Leftrightarrow \;\; & \exists i \;\; p^* \times g_i \times a - c^* \times g_i \times b + \lfloor d_i/g_i \rfloor \times g_i + d_i \bmod g_i < c^* \times g_i \\
\Leftrightarrow \;\; & \exists i \;\; p^* \times a - c^* \times b + \lfloor d_i/g_i \rfloor \leq c^* - 1 \\
\Leftrightarrow \;\; & p^* \times a - c^* \times b + d^* \leq c^* - 1 \\
\Leftrightarrow \;\; & p^* \times a - c^* \times b + d^* = tok_{G^*}(S, e^*, k_{src}-1) < c^*
\end{aligned}
\tag{7.4}
$$

As with Lemma 7.11, the *if* direction can be proved in a similar manner by applying the same derivations in reverse order and based on the reverse directions in both Equation (7.3) and Equation (7.4). □

The following corollary follows from Property 7.13 and Lemma 7.14.

Corollary 7.15. *A minimum buffer schedule for a consistent, acyclic, two-actor SDF graph can be obtained by computing an SASAP schedule for its primitive form.*

The following property follows from Equation (7.2) and Definition 7.10 and relates the buffer activity in an acyclic, two-actor SDF graph to that of its primitive form.

Property 7.16. *Suppose S is a valid schedule for a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ and for the primitive form $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$ of G. Then for every edge $e_i \in E$, and for every firing index k, $tok_G(S, e_i, k) = tok_{G^*}(S, e^*, k) \times g_i + d_i - d^* \times g_i$, where $d_i = del(e_i)$, $g_i = \gcd(prd(e_i), cns(e_i))$, and $d^* = \min_{e_i \in E}(\lfloor d_i/g_i \rfloor)$.*

Theorem 7.17. *For a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$, the minimum buffer requirement for an edge $e_i \in E$ is $p_i + c_i - g_i + d_i - d^* \times g_i$ if $0 \leq d^* \leq p^* + c^* - 1$, and is d_i otherwise. Here, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, $c^* = c_i/g_i$, and $d^* = \min_{e_i \in E}(\lfloor d_i/g_i \rfloor)$.*

Proof. For a consistent, single-edge, two-actor SDF graph $(\{v_{src}, v_{snk}\}, \{e = (v_{src}, v_{snk})\})$, Bhattacharyya at al. [7] have proved that the minimum buffer requirement for e is $p + c - g + d \bmod g$ if $0 \leq d \leq p + c - g$, and is d otherwise, where $p = prd(e)$, $c = cns(e)$, $d = del(e)$, and $g = \gcd(p, c)$. As a result, the minimum buffer requirement for e^* is $p^* + c^* - 1$ if $0 \leq d^* \leq p^* + c^* - 1$, and is d^* otherwise. From Lemma 7.14, Property 7.13, Property 7.16, and the minimum buffer requirement for e^*, the proof is complete. □

Theorem 7.17 presents a constant-time minimum buffer computation for any consistent, acyclic, two-actor SDF graph, and it is used in our adapted form of DPPO to compute buffer requirements for each nested two-actor subgraph $G_{i,j,k} = (\{\alpha_{i,k}, \alpha_{k+1,j}\}, E'_{i,j,k})$ as described in Section 7.3.8.

In order to build an overall schedule from a nested two-actor DPPO hierar-

chy, we compute an optimal buffer schedule for each two-actor subgraph. Based on Corollary 7.15, we have developed the BOTAS (buffer-optimal two-actor scheduling) algorithm. This algorithm is shown in Figure 7.7. The BOTAS algorithm computes a minimum buffer schedule for a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ by constructing an SASAP schedule for its primitive form.

In Figure 7.7, we first compute p^*, c^*, and d^* of G to construct the primitive form $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$. Then in lines 6-19, we compute two sequences of scheduling components $A_1, A_2, \ldots, A_I$ and $B_1, B_2, \ldots, B_I$, where I denotes the iteration i that ends the while loop. Table 7.1 illustrates how to compute the sets of scheduling components for $p^* = 7$ and $c^* = 5$. For convenient schedule loop representation in Figure 7.7, we define the expression $(k \times L)$ for a positive integer k and a schedule loop $L = (n\ T_1 T_2 \cdots T_m)$ as a new schedule loop with the same loop body $T_1 T_2 \cdots T_m$ and the new iteration count $k \times n$, i.e., $(k \times L) = (k \times n\ T_1 T_2 \cdots T_m)$.

From the results of this computation, we construct an SASAP schedule S for G^*. If the initial token population $d^* = 0$, S can be immediately built from A_I and B_I by line 22. Otherwise, in lines 25-38, we first use the scheduling components A_i and B_i from $i = 1$ to I to consume initial tokens until either the token population d on e^* is 0 or we exhaust execution of v_{snk}. Then in lines 39-49, we use the scheduling components from $i = I$ to 1 to make up the remaining firings of v_{src} and v_{snk}, and bring the token population back to its initial state d^*. Table 7.1 illustrates how to compute SASAP schedules for $d^* = 0$, 6, and 12.

From the BOTAS algorithm, we can directly derive the following properties.

BOTAS($G \equiv (\{v_{src}, v_{snk}\}, E)$) /*The input G is a consistent acyclic two-actor SDF graph*/

1 $p^* \leftarrow prd(e_1)/\gcd(prd(e_1), cns(e_1))$
2 $c^* \leftarrow cns(e_1)/\gcd(prd(e_1), cns(e_1))$
3 $d^* \leftarrow \min_{e_i \in E}(\lfloor del(e_i)/\gcd(prd(e_i), cns(e_i))\rfloor)$
4 $A_1 \leftarrow v_{src}$, $B_1 \leftarrow v_{snk}$, $p_1 \leftarrow p^*$, $c_1 \leftarrow c^*$, $m_{A_1} \leftarrow 1$, $n_{A_1} \leftarrow 0$, $m_{B_1} \leftarrow 0$, $n_{B_1} \leftarrow 1$
5 $i \leftarrow 1$
6 **while !**($p_i \bmod c_i = 0$ **or** $c_i \bmod p_i = 0$)
7 **if** $p_i > c_i$
8 $A_{i+1} \leftarrow (1\ A_i\ (\lfloor p_i/c_i \rfloor \times B_i))$, $p_{i+1} \leftarrow p_i \bmod c_i$
9 $m_{A_{i+1}} \leftarrow m_{A_i} + \lfloor p_i/c_i \rfloor \times m_{B_i}$, $n_{A_{i+1}} \leftarrow n_{A_i} + \lfloor p_i/c_i \rfloor \times n_{B_i}$
10 $B_{i+1} \leftarrow (1\ A_i\ (\lceil p_i/c_i \rceil \times B_i))$, $c_{i+1} \leftarrow c_i - p_i \bmod c_i$
11 $m_{B_{i+1}} \leftarrow m_{A_i} + \lceil p_i/c_i \rceil \times m_{B_i}$, $n_{B_{i+1}} \leftarrow n_{A_i} + \lceil p_i/c_i \rceil \times n_{B_i}$
12 **else**
13 $A_{i+1} \leftarrow (1\ (\lceil c_i/p_i \rceil \times A_i)\ B_i)$, $p_{i+1} \leftarrow p_i - c_i \bmod p_i$
14 $m_{A_{i+1}} \leftarrow \lceil c_i/p_i \rceil \times m_{A_i} + m_{B_i}$, $n_{A_{i+1}} \leftarrow \lceil c_i/p_i \rceil \times n_{A_i} + n_{B_i}$
15 $B_{i+1} \leftarrow (1\ (\lfloor c_i/p_i \rfloor \times A_i)\ B_i)$, $c_{i+1} \leftarrow c_i \bmod p_i$
16 $m_{B_{i+1}} \leftarrow \lfloor c_i/p_i \rfloor \times m_{A_i} + m_{B_i}$, $n_{B_{i+1}} \leftarrow \lfloor c_i/p_i \rfloor \times n_{A_i} + n_{B_i}$
17 **end**
18 $i \leftarrow i + 1$
19 **end**
20 $I \leftarrow i$
21 **if** $d^* = 0$
22 **if** $p_I > c_I$ $S \leftarrow A_I\ (p_I/c_I \times B_I)$ **else** $S \leftarrow (c_I/p_I \times A_I)\ B_I$ **end**
23 **return** S
24 **else**
25 $S \leftarrow \varnothing$, $d \leftarrow d^*$, $m \leftarrow c^*$, $n \leftarrow p^*$
26 **for** i from 1 to I
27 **if** $d \geq c_i$
28 **if** $(x \leftarrow \lfloor d/c_i \rfloor) \times n_{B_i} > n$ **break** **end**
29 $S \leftarrow S\ (x \times B_i)$, $d \leftarrow d \bmod c_i$, $m \leftarrow m - x \times m_{B_i}$, $n \leftarrow n - x \times n_{B_i}$
30 **if** $d = 0$ **break** **end**
31 **end**
32 **if** $d \geq p_i$
33 **if** $(x \leftarrow \lceil (c_i - d)/p_i \rceil) \times n_{A_i} + n_{B_i} > n$ **break** **end**
34 $S \leftarrow S\ (x \times A_i)\ B_i$, $d \leftarrow d + x \times p_i - c_i$
35 $m \leftarrow m - x \times m_{A_i} - m_{B_i}$, $n \leftarrow n - x \times n_{A_i} - n_{B_i}$
36 **if** $d = 0$ **break** **end**
37 **end**
38 **end**
39 **for** i from I to 1
40 $x \leftarrow \min(\lfloor m/m_{A_i} \rfloor, \lfloor n/n_{A_i} \rfloor, \lceil (c_i - d)/p_i \rceil)$
41 **if** $x \geq 1$
42 $S \leftarrow S\ (x \times A_i)$, $d \leftarrow d + x \times p_i$, $m \leftarrow m - x \times m_{A_i}$, $n \leftarrow n - x \times n_{A_i}$
43 **end**
44 $x \leftarrow \min(\lfloor m/m_{B_i} \rfloor, \lfloor n/n_{B_i} \rfloor, \lfloor d/c_i \rfloor)$
45 **if** $x \geq 1$
46 $S \leftarrow S\ (x \times B_i)$, $d \leftarrow d - x \times c_i$, $m \leftarrow m - x \times m_{B_i}$, $n \leftarrow n - x \times n_{B_i}$
47 **end**
48 **end**
49 $S \leftarrow S\ (m \times A_1)$
50 **return** S
51 **end**

Figure 7.7: Buffer-optimal two-actor scheduling (BOTAS) algorithm.

Table 7.1: Demonstration of buffer-optimal two-actor scheduling for $p^* = 7$, $c^* = 5$, and $d^* = 0, 6, 12$, where $a = v_{src}$ and $b = v_{snk}$.

Computing scheduling components for $p^* = 7$ $c^* = 5$					
lines	i	A_i	B_i	p_i	c_i
4-20	1	a	b	7	5
	2	$(1\ A_1\ B_1)$	$(1\ A_1\ (2\ B_1))$	2	3
	3	$(1\ (2\ A_2)\ B_2)$	$(1\ A_2\ B_2)$	1	1

Two-actor scheduling for $d^* = 0$	
lines	S
22-23	$A_3B_3 = (1(2ab)(1a(2b)))(1(1ab)(1a(2b)))$

Two-actor scheduling for $d^* = 6$		
lines	i	S
25-38	1	$B_1 = b$
	3	$S\ B_3 = b(1(1ab)(1a(2b)))$
39-48	2	$S\ (2A_2) = b(1(1ab)(1a(2b)))(2ab)$
	1	$S\ A_1B_1 = b(1(1ab)(1a(2b)))(2ab)ab$

Two-actor scheduling for $d^* = 12$		
lines	i	S
25-38	1	$(2B_1) = (2b)$
	2	$S\ A_2B_2 = (2b)(1ab)(1a(2b))$
39-48	2	$S\ A_2 = (2b)(1ab)(1a(2b))(1ab)$
	1	$S\ A_1B_1 = (2b)(1ab)(1a(2b))(1ab)ab$
49	N/A	$S\ A_1 = (2b)(1ab)(1a(2b))(1ab)aba$

Property 7.18. *In the BOTAS algorithm, each A_i produces p_i tokens and fires v_{snk} ASAP, and each B_i consumes c_i tokens and fires v_{snk} ASAP whenever there are c_i tokens.*

Property 7.19. *In the BOTAS algorithm, for every $i \in \{1, 2, \ldots, I-1\}$, $p_{i+1} + c_{i+1} = \min(p_i, c_i)$.*

The following property is derived from the BOTAS algorithm, Euclid's algorithm for computation of greatest common divisors, and mathematical induction.

Property 7.20. *In the BOTAS algorithm, for every $i \in \{1, 2, \ldots, I\}$, $\gcd(p_i, c_i) = 1$.*

Proof. Initially, $\gcd(p_1, c_1) = \gcd(p^*, c^*) = 1$. Suppose in an iteration $i > 1$, $\gcd(p_i, c_i) = 1$. By Equation (7.5) when $p_i > c_i$, and by Equation (7.6) when $p_i < c_i$, we can derive that $\gcd(p_{i+1}, c_{i+1}) = 1$.

$$\begin{aligned} \gcd(p_i, c_i) = 1 \;&\Rightarrow\; \gcd(c_i, p_i \mod c_i) = 1 \text{ and } \gcd(c_i, c_i - p_i \mod c_i) = 1 \\ &\Rightarrow\; \gcd(p_{i+1}, c_{i+1}) = 1 \end{aligned} \tag{7.5}$$

$$\begin{aligned} \gcd(c_i, p_i) = 1 \;&\Rightarrow\; \gcd(p_i, c_i \mod p_i) = 1 \text{ and } \gcd(p_i, p_i - c_i \mod p_i) = 1 \\ &\Rightarrow\; \gcd(c_{i+1}, p_{i+1}) = 1 \end{aligned} \tag{7.6}$$

By mathematical induction, $\forall i \in \{1, 2, \ldots, I\}$, $\gcd(p_i, c_i) = 1$. □

Directly from Property 7.20, we can derive the following termination property.

Property 7.21. *In the BOTAS algorithm, the while loop in line 6 terminates when either $p_i = 1$ or $c_i = 1$.*

The following lemma determines a bound on the number of iterations I of the while loop in line 6. In practical cases, I is usually much smaller than the bound.

Lemma 7.22. *In the BOTAS algorithm, the iteration number I that terminates the while loop in line 6 is bounded by* $\log_2 \min(p^*, c^*)$.

Proof. From Property 7.19, it follows that $\min(p_{i+1}, c_{i+1}) \leq \min(p_i, c_i)/2$. Because the while loop ends when $p_i = 1$ or $c_i = 1$ (Property 7.21), it takes at most $\log_2 \min(p_1, c_1)$ iterations to achieve $p_i = 1$ or $c_i = 1$. Therefore, the iteration $i = I$ that terminates the while loop is bounded by $\log_2 \min(p^*, c^*)$. □

Finally, we establish the correctness, optimality, and complexity of the BOTAS algorithm in Theorem 7.23, Property 7.24, and Property 7.25.

Theorem 7.23. *In the BOTAS algorithm, suppose there are no initial tokens on edge e^* ($d^* = 0$), and define the schedule $S = A_I\ (p_I/c_I \times B_I)$ if $p_I > c_I$, and $S = (c_I/p_I \times A_I)B_I$, otherwise (line 22). Then S is an SASAP schedule for G^*.*

Proof. From Property 7.18, it follows that S fires v_{snk} ASAP. We then show that S fires v_{src} c^* times and v_{snk} p^* times to prove that S is an SASAP schedule.

For $p_I > c_I$ and $p_{I-1} > c_{I-1}$, because $p_I = p_{I-1} \bmod c_{I-1}$, $c_I = c_{I-1} - p_{I-1} \bmod c_{I-1} = 1$, $\lceil p_{I-1}/c_{I-1} \rceil = \lfloor p_{I-1}/c_{I-1} \rfloor + 1$, $S = A_I\ (p_I/c_I\ B_I)$, $A_I = (1\ A_{I-1}\ (\lfloor p_{I-1}/c_{I-1} \rfloor\ B_{I-1}))$, and $B_I = (1\ A_{I-1}\ (\lceil p_{I-1}/c_{I-1} \rceil\ B_{I-1}))$, we can derive that S fires A_{I-1}

$$1 + p_I/c_I = 1 + c_{I-1} - 1 = c_{I-1} \text{ times},$$

and S fires B_{I-1}

$$\lfloor p_{I-1}/c_{I-1} \rfloor + p_I/c_I \times \lceil p_{I-1}/c_{I-1} \rceil$$

$$= \lfloor p_{I-1}/c_{I-1} \rfloor + (c_{I-1}-1) \times \lfloor p_{I-1}/c_{I-1} \rfloor + p_{I-1} \bmod c_{I-1} = p_{I-1} \text{ times.}$$

By using a similar approach, we can derive that S fires A_{I-1} c_{I-1} times and fires B_{I-1} p_{I-1} times for the following cases: (a) $p_I > c_I$ and $p_{I-1} < c_{I-1}$; (b) $p_I \leq c_I$ and $p_{I-1} > c_{I-1}$; and (c) $p_I \leq c_I$ and $p_{I-1} < c_{I-1}$.

Now, suppose S fires A_i c_i times and fires B_i p_i times for some $i \in \{2, 3, \ldots, I-1\}$. Then if $p_{i-1} > c_{i-1}$, we can derive that S fires A_{i-1}

$$(c_{i-1} - p_{i-1} \bmod c_{i-1}) + (p_{i-1} \bmod c_{i-1}) = c_{i-1} \text{ times,}$$

and S fires B_{i-1}

$$(c_{i-1} - p_{i-1} \bmod c_{i-1}) \times \lfloor p_{i-1}/c_{i-1} \rfloor + (p_{i-1} \bmod c_{i-1}) \times \lceil p_{i-1}/c_{i-1} \rceil$$

$$= c_{i-1} \times \lfloor p_{i-1}/c_{i-1} \rfloor + p_{i-1} \bmod c_{i-1} = p_{i-1} \text{ times,}$$

because $A_i = (1\ A_{i-1}\ (\lfloor p_{i-1}/c_{i-1} \rfloor\ B_{i-1}))$, $B_i = (1\ A_{i-1}\ (\lceil p_{i-1}/c_{i-1} \rceil\ B_{i-1}))$, $p_i = p_{i-1} \bmod c_{i-1}$, and $c_i = c_{i-1} - p_{i-1} \bmod c_{i-1}$.

In a similar way, when $p_{i-1} < c_{i-1}$, we can derive that S fires A_{i-1} c_{i-1} times and fires B_{i-1} p_{i-1} times.

As a result, if S fires A_i c_i times and fires B_i p_i times for some $i \in \{2, 3, \ldots, I-1\}$, then S fires A_{i-1} c_{i-1} times and fires B_{i-1} p_{i-1} times. Because we have proved that S fires A_{I-1} c_{I-1} times and fires B_{I-1} p_{I-1} times, we can conclude by mathematical induction that S fires A_i c_i times and fires B_i p_i times for every $i \in \{1, 2, \ldots, I-1\}$. Taking $i = 1$, we have that S fires v_{src} c^* times and fires v_{snk} p^* times. □

Using demand-driven analysis on the state of $e*$ (i.e., d in BOTAS) and the

numbers of remaining firings of v_{src} and v_{snk} (i.e., m and n in BOTAS), the following result can be shown. This result, together with Theorem 7.23, establishes the correctness of the BOTAS algorithm.

Property 7.24. *Suppose that there are d^* initial tokens on edge e^* ($d^* > 0$). Then the BOTAS algorithm in lines 25-49 constructs an SASAP schedule for G^*.*

Property 7.25. *The complexity of the BOTAS algorithm is* $\mathrm{O}(\log_2 \min(p^*, c^*))$.

Proof. From Lemma 7.22, the iteration I that ends the while loop in line 6 is bounded by $\log_2 \min(p^*, c^*)$. The first for loop (lines 26-38) and the second for loop (lines 39-48) are both bounded by I. All other operations can be implemented in constant time. As a result, the complexity of the BOTAS algorithm is $\mathrm{O}(\log_2 \min(p^*, c^*))$. □

7.3.10 Buffering for Cycle-Broken Edges

From Section 7.3.2, even though scheduling acyclic graphs that emerge from the LIAF decomposition process without considering the removed inter-iteration edges never violates data precedence constraints, buffer sizes of the removed edges should still be properly computed based on the scheduling results. Otherwise, during execution, the graph may deadlock or produce memory corruption due to buffer overflow. In this section, we analyze buffer bounds for inter-iteration edges that are removed by cycle-breaking.

Our analysis here assumes that the acyclic graphs that emerge from LIAF are scheduled based on *R-schedule* or *R-hierarchy*. A valid single appearance schedule S

is an *R-schedule* if S and each of the nested schedule loops in S has *either* 1) a single iterand, and this single iterand is an actor, *or* 2) exactly two iterands, and these two iterands are schedule loops having coprime iteration counts [7] . In general, an R-schedule can be viewed as providing a single appearance, minimal periodic schedule for each two-actor graph in the *R-hierarchy*. Here by R-hierarchy, we mean the *nested two-actor cluster hierarchy* that is obtained from the looped binary structure in the R-schedule.

A variety of single appearance scheduling techniques fall into the domain of R-schedules — for example, APGAN [7], DPPO [7], and RPMC [58]. Furthermore, the recursive procedure call based technique [45] and the buffer-optimal two-actor scheduling algorithm (see Section 7.3.9) also work on recursive, multiple appearance schedules of each two-actor graph in the R-hierarchy.

Analysis of buffer bounds on the removed inter-iteration edges can be performed by studying the configuration of the removed edges in the R-hierarchy. Suppose that we are given a consistent, loosely interdependent, strongly connected SDF graph G. Suppose also that the CYCLE-BREAKING algorithm (see Section 7.3.3) removes a subset of inter-iteration edges E' from G, and suppose G' is the acyclic SDF graph that is constructed by clustering the SCCs of the resulting graph G. As described earlier, we assume that R-schedule or R-hierarchy based techniques are applied to scheduling G'. Then we have the following observations: 1) By joint analysis of G' and the given R-schedule, a R-hierarchy H can always be constructed such that each two-actor graph in H is consistent and acyclic, and the order of the leaf actors encountered in depth-first, source-to-sink traversal of H gives a topological

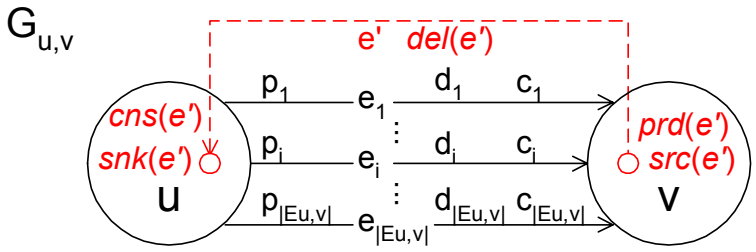


Figure 7.8: Presence of cycle-broken edge in the two-actor graph.

sort of G'. 2) According to Theorem 7.3, a removed inter-iteration edge $e' \in E'$ must connect a succeeding actor (or SCC supernode) to the first actor (or SCC supernode) in the topological sort. 3) The final schedule S' of G' can be decomposed such that for each two-actor SDF subgraph in H, there exists a corresponding minimal periodic sub-schedule (single appearance or multiple appearance) in S'.

Based on 1), 2), and 3), analysis of buffer bounds on a removed inter-iteration edge e' can be performed in the *unique* two-actor graph in H, $G_{u,v} = (\{u, v\}, E_{u,v} = \{e \mid src(e) = u \text{ and } snk(e) = v\})$, such that $src(e')$ is in the v-cluster and $snk(e')$ is in the u-cluster. In other words, there exists a unique, consistent, acyclic, two-actor SDF graph $G_{u,v}$ in H such that the presence of e' is in the reverse direction across the two actors in $G_{u,v}$. Figure 7.8 shows a general form of such configuration, where for each $e_i \in E_{u,v}$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$.

The following theorem pertains to the buffer bounds on the removed inter-iteration edges.

Theorem 7.26. *Suppose that we are given a consistent, loosely interdependent, strongly connected SDF graph G. Suppose G' is the acyclic SDF graph that is constructed by applying the CYCLE-BREAKING algorithm on G and clustering the*

SCCs of the resulting graph G. Suppose H is the R-hierarchy in scheduling G'. Suppose that e' is an inter-iteration edge that is removed by the CYCLE-BREAKING algorithm. Suppose $G_{u,v} = (\{u, v\}, E_{u,v} = \{e \mid src(e) = u \text{ and } snk(e) = v\})$ is the consistent, acyclic, two-actor SDF graph in H such that $src(e')$ is in the v-cluster and $snk(e')$ is in the u-cluster. Then the buffer size required for e' is bounded by

$$\begin{cases} del(e') + g \times d^* & \text{if } d^* \leq p^* \times c^* \\ del(e') + g \times p^* \times c^* & \text{if } d^* > p^* \times c^*. \end{cases} \tag{7.7}$$

Here, p^*, c^*, and d^* are the primitive production rate, primitive consumption rate, and primitive delay of $G_{u,v}$, respectively; and in addition, $g = \gcd(p, c)$, $p = prd(e') \times \boldsymbol{q}_G[src(e')]/g_v$, $c = cns(e') \times \boldsymbol{q}_G[snk(e')]/g_u$, $g_u = \gcd_{\alpha \in u-\text{cluster}}(\boldsymbol{q}_G[\alpha])$, and $g_v = \gcd_{\alpha \in v-\text{cluster}}(\boldsymbol{q}_G[\alpha])$.

Proof. Based on Definition 7.10 and Lemma 7.11, analysis of $G_{u,v}$ in Figure 7.8 is equivalent to analysis of its primitive form $G^*_{u,v} = (\{u, v\}, \{e^* = (u, v)\})$ in Figure 7.9, where for each $e_i \in E_{u,v}$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, and $c^* = c_i/g_i$; for e^*, $prd(e^*) = p^*$, $cns(e^*) = c^*$, $\gcd(p^*, c^*) = 1$, and $del(e^*) = d^* = \min_{e_i \in E_{u,v}}(\lfloor d_i/g_i \rfloor)$.

Furthermore, because of the properties of SDF clustering, we can derive that 1) $p^* \times g_u = c^* \times g_v$, 2) execution of u consists of executing $snk(e')$ for $\boldsymbol{q}_G[snk(e')]/g_u$ times, and 3) execution of v consists of executing $src(e')$ for $\boldsymbol{q}_G[src(e')]/g_v$ times. As a result, we can transform e' in Figure 7.8 to an equivalent edge e in Figure 7.9 such that $src(e) = v$, $snk(e) = u$, $prd(e) = p$, $cns(e) = c$, and $del(e) = d = del(e')$. Note that adding e to $G^*_{u,v}$ preserves consistency because 1) $p^*/c^* = c/p$ — this is

because of the balance equation on e':

$$prd(e') \times \boldsymbol{q}_G[src(e')] = cns(e') \times \boldsymbol{q}_G[snk(e')], \tag{7.8}$$

and 2) d is large enough for the consumption requirements of u for a complete iteration of $G^*_{u,v}$ — this is because e' is an inter-iteration edge for G so that

$$d = del(e') \geq cns(e') \times \boldsymbol{q}_G[snk(e')] = c \times g_u \geq c \times c^* \tag{7.9}$$

Based on Lemma 7.11, suppose S is any valid minimal periodic schedule for $G_{u,v}$ as well as $G^*_{u,v}$. According to Equation (7.2), we can derive that

$$tok_{G^*_{u,v}}(S, e, k) = \tau(S, v, k) \times p - \tau(S, u, k) \times c + d \tag{7.10}$$

and

$$tok_{G^*_{u,v}}(S, e^*, k) = \tau(S, u, k) \times p^* - \tau(S, v, k) \times c^* + d^* \tag{7.11}$$

Then, we can derive the following equation based on Equation (7.10) and Equation (7.11).

$$tok_{G^*_{u,v}}(S, e, k) = d + g \times (d^* - tok_{G^*_{u,v}}(S, e^*, k)) \tag{7.12}$$

Because S is a valid minimal periodic schedule, for any firing index k, we can derive that

$$\begin{aligned} &tok_{G^*_{u,v}}(S, e^*, k) \geq 0 && \text{if } d^* \leq p^* \times c^*, \\ &tok_{G^*_{u,v}}(S, e^*, k) \geq d^* - p^* \times c^* && \text{if } d^* > p^* \times c^*. \end{aligned} \tag{7.13}$$

Finally, substituting Equation (7.13) into Equation (7.12) gives us

$$\begin{aligned} &tok_{G^*_{u,v}}(S, e, k) \leq d + g \times d^* && \text{if } d^* \leq p^* \times c^*, \\ &tok_{G^*_{u,v}}(S, e, k) \leq d + g \times p^* \times c^* && \text{if } d^* > p^* \times c^*. \end{aligned} \tag{7.14}$$

The proof is complete. □

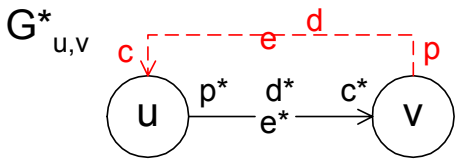


Figure 7.9: Presence of cycle-broken edge in the primitive two-actor graph.

7.3.11 Schedule Representation

In our implementation of SOS, we have also given attention to efficient management of the data structure that stores the computed schedule. In this data structure, each schedule loop is created only once, and multiple references to a schedule loop across the overall schedule are implemented as pointers to the single version. We apply this concept in the construction of scheduling components in the two-actor algorithm as well as in supernode/sub-schedule replacements across cluster boundaries. This implementation can significantly reduce the memory requirement for representing the overall schedule, and is more suited to the simulation-based context of this work than the procedure-call based implementation format of [45], which is more suited to software synthesis.

7.4 Overall Integration

The overall integration of component algorithms in SOS is illustrated in Figure 7.1. A major contribution of this work is the selection, adaptation, and integration of these algorithms — along with development of associated theory and analysis — into a complete simulation environment for the novel constraints associated with simulating critical SDF graphs. In fact, the complexity involved in the overall SOS

approach is dominated by the complexity of scheduling subgraphs that it isolates in its top-down process of LIAF- and SRC-based decomposition. For this reason, we are able to apply the intensive APGAN, DPPO, and buffer-optimal two-actor scheduling algorithms in SOS without major degradation in simulation performance. This is beneficial because these intensive techniques provide significant reductions in the total buffer requirement.

Figure 7.10 presents an example to illustrate SOS. Given a connected, consistent SDF graph (e.g., Figure 7.10.(a)), SOS first applies LIAF (Section 7.3.2) to decompose all strongly connected components in order to derive an acyclic SDF graph (as illustrated in Figure 7.10.(d)) and break cycles for strongly connected subgraphs (as illustrated in Figure 7.10.(b) and Figure 7.10.(c)). If a subgraph is loosely interdependent, LIAF is applied recursively to derive a schedule for the subgraph (e.g., $(1F(1(7I)J))$ for Figure 7.10.(b) and $(1(1(2G)H)(1GH))$ for Figure 7.10.(c)).

For the acyclic graph, SOS applies SRC (Section 7.3.5) to isolate single-rate subgraphs, and reduce the acyclic graph into a smaller multirate version. This is illustrated in Figure 7.10.(g). For single-rate subgraphs (e.g., Figure 7.10.(e) and Figure 7.10.(f)), SOS efficiently computes schedules (e.g., $(1ABD)$ for Figure 7.10.(e) and $(1ZC)$ for Figure 7.10.(f)) by the flat scheduling approach (Section 7.3.6).

After SRC, SOS uses APGAN (Section 7.3.7) to obtain a buffer-efficient topological sort (e.g., Figure 7.10.(h)) for the multirate, acyclic graph. Then from the topological sort, SOS applies DPPO (Section 7.3.8) to construct a buffer-efficient two-actor hierarchy. This is illustrated in Figure 7.10.(i). Finally, SOS computes a

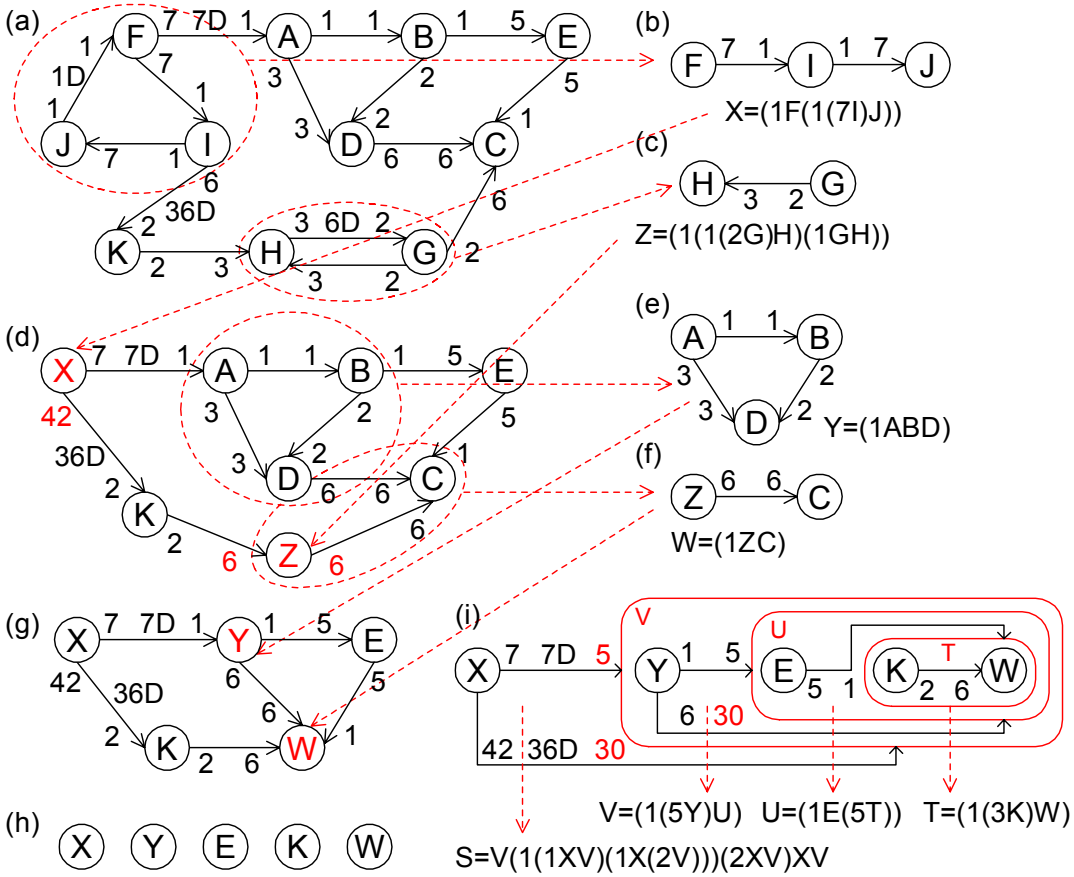


Figure 7.10: SOS scheduling example.

Table 7.2: Characteristics of wireless communication designs.

Design #	Description	number of actors	number of edges (single-/multirate)	Multirate Complexity
1	3GPP Uplink Source	82	133 (101 / 32)	1.86 E6
2	3GPP Downlink Source	179	236 (194 / 42)	1.10 E6
3	Bluetooth Packets	104	107 (97 / 10)	808
4	802.16e Source Constellation	71	73 (49 / 24)	9.95 E6
5	CDMA2000 Digital Distortion	707	855 (805 / 50)	3.83 E6
6	XM Radio	269	293 (245 / 48)	5.43 E6
7	Edge Signal Source	186	222 (192 / 30)	36.36 E6
8	Digital TV	114	126 (74 / 52)	1.37 E6
9	WiMax Downlink Source	368	389 (276 / 113)	73191

buffer-optimal schedule for each two-actor subgraph based on the two-actor scheduling algorithm (Section 7.3.9) — for example, $(1(3K)W)$ for the two-actor subgraph $\{K, W\}$, $(1E(5T))$ for the subgraph $\{E, T\}$, $(1(5Y)U)$ for the subgraph $\{Y, U\}$, and $V(1(1XV)(1X(2V)))(2XV)XV$ for the top-level two-actor graph $\{X, V\}$. An overall schedule is then obtained by traversing the constructed hierarchies and replacing supernodes with the corresponding sub-schedules.

7.5 Simulation Results

We have implemented and integrated the simulation-oriented scheduler in Agilent ADS [67]. Here, we demonstrate our simulation-oriented scheduler by scheduling and simulating state-of-the-art wireless communication systems in ADS. However, the design of SOS is not specific to ADS, and the techniques presented in this chapter can be generally implemented in any simulation tool that incorporates SDF semantics.

The experimental platform is a PC with 1GHz CPU and 1GB memory. In our experiments, we include 9 wireless communication designs from Agilent Technologies in the following standards: 3GPP (WCDMA3G), Bluetooth, 802.16e (WiMax), CDMA 2000, XM radio, EDGE, and Digital TV. Table 7.2 presents characteristics of the 9 designs, including the numbers of actors, numbers of edges (single-rate/multirate), and approximate multirate complexities. These designs contain from several tens to hundreds of actors and edges, and possess very high multirate complexities. In particular, the multirate complexities in designs 1, 2, 4, 5, 6, 7, and 8 are in the range of millions.

We simulate the 9 designs with our simulation-oriented scheduler (SOS), and the present default cluster-loop scheduler (CLS) in ADS. The simulation results of CLS, SOS, and the performance ratio (CLS/SOS) are shown in three tables: Table 7.3 presents the total buffer requirements for SDF edges (in number of tokens); Table 7.4 presents the average scheduling time of ten runs (in seconds); and Table 7.5 presents the average total simulation time of ten runs (in seconds). As shown in these tables, SOS outperforms CLS in almost all designs in terms of memory requirements, scheduling time, and total simulation time (except design 3 and 9, which are comparable due to their relatively small multirate complexities). In particular, SOS is effective in reducing buffer requirements within short scheduling time. For design 9, CLS requires less scheduling time because of its capabilities as a fast heuristic. However, for design 2, it requires a very long scheduling time due to its heavy dependence on classical SDF scheduling. CLS fails in design 5 due to an out-of-memory problem during scheduling (OOM-sc), and also fails in designs 1, 4, 6, and 7 due to

Table 7.3: Total buffer requirements (tokens).

Design	CLS	SOS	Ratio (CLS/SOS)
1	50445629	229119	220
2	9073282	43247	210
3	3090	3090	1
4	89428569	669273	134
5	OOM-sc	9957292	N/A
6	48212523	5385031	9
7	1870248382	451862	4139
8	8257858	1976318	4
9	1834606	1832926	1

Table 7.4: Average scheduling time (seconds).

Design	CLS	SOS	Ratio (CLS/SOS)
1	0.08	0.08	1.00
2	279.11	0.16	1744.44
3	0.06	0.06	1.00
4	0.49	0.45	1.09
5	OOM-sc	13.50	N/A
6	10.72	0.67	16.00
7	0.92	0.87	1.06
8	2.73	0.53	5.15
9	3.59	9.98	0.36

out-of-memory problems in buffer allocation (OOM-ba). With SOS, we are able to simulate these heavily multirate designs.

7.6 Conclusion

In this chapter, we have introduced and illustrated the challenges in scheduling large-scale, highly multirate synchronous dataflow (SDF) graphs for simulation tools that incorporate SDF semantics. We have defined critical SDF graphs as

Table 7.5: Average total simulation time (seconds).

Design	CLS	SOS	Ratio (CLS/SOS)
1	OOM-ba	7.12	N/A
2	349.33	55.31	6.32
3	930.56	876.72	1.06
4	OOM-ba	203.95	N/A
5	OOM-sc	2534.06	N/A
6	OOM-ba	406.86	N/A
7	OOM-ba	28940.77	N/A
8	636.63	415.40	1.53
9	1566.92	1542.39	1.02

an important class of graphs that must be taken carefully into consideration when designing such tools for modeling and simulating modern large-scale and heavily multirate communication and signal processing systems. We have then presented the simulation-oriented scheduler (SOS). SOS integrates several existing and newly-developed graph decomposition and scheduling techniques in a strategic way for joint run-time and memory minimization in simulating critical SDF graphs. We have demonstrated the efficiency of our scheduler by simulating practical, large-scale, and highly multirate wireless communication designs.

Chapter 8

Multithreaded Simulation of Synchronous Dataflow Graphs

For system simulation, synchronous dataflow (SDF) has been widely used as a core model of computation in design tools for digital communication and signal processing systems. The traditional approach for simulating SDF graphs is to compute and execute static schedules in single-processor desktop environments. Nowadays, however, multi-core processors are increasingly popular desktop platforms for their potential performance improvements through on-chip, thread-level parallelism. Without novel scheduling and simulation techniques that explicitly explore thread-level parallelism for executing SDF graphs, current design tools gain only minimal performance improvements on multi-core platforms. In this chapter, we present a new multithreaded simulation scheduler, called MSS, to provide simulation runtime speed-up for executing SDF graphs on multi-core processors. MSS strategically integrates graph clustering, intra-cluster scheduling, actor vectorization, and inter-cluster buffering techniques to construct inter-thread communication (ITC) graphs at compile-time. MSS then applies efficient synchronization and dynamic scheduling techniques at runtime for executing ITC graphs in multithreaded environments. We have implemented MSS in the Advanced Design System (ADS) from Agilent Technologies. On an Intel dual-core, hyper-threading (4 processing units) processor, our results from this implementation demonstrate up to 3.5 times speed-up in simulating

modern wireless communication systems (e.g., WCDMA3G, CDMA 2000, WiMax, EDGE, and Digital TV).

8.1 Introduction

Nowadays, multi-core processors are increasingly popular desktop platforms for their potential performance improvements through on-chip, thread-level parallelism. This type of on-chip, thread-level parallelism can be further categorized into chip-level multiprocessing (CMP) [29] (e.g., dual-core or quad-core CPUs from Intel or AMD) and simultaneous multithreading (SMT) [20] (e.g., hyper-threading CPUs from Intel). However, without novel scheduling and simulation techniques that explicitly explore thread-level parallelism for executing SDF graphs, current EDA tools gain only minimal performance improvements from these new sets of processors. This is largely due to the sequential (single-thread) SDF execution semantics that underlies these tools.

In general, the design space of scheduling dataflow graphs for parallel computation is highly complex. For synthesis of DSP systems onto embedded multiprocessors, scheduling and synchronization techniques in the domain of homogeneous synchronous dataflow (HSDF) [75] have been extensively studied in the literature (see Chapter 3). Detailed definitions and background related to SDF and HSDF are given in Section 8.2.

Based on a scheduling taxonomy presented in [49], scheduling HSDF graphs for multiprocessor implementation consists of the following tasks: *assignment* —

assigning actors (individual dataflow tasks) to processors, *ordering* — ordering execution of actors on each processor, and *timing* — determining when each actor executes.

In this chapter, we focus on multithreaded simulation of SDF graphs, which is a new research area motivated by the increasing popularity of on-chip, thread-level parallel computation. Our target simulation platforms are current multi-core processors, and the objective is to *speed up* simulation runtime (including time for scheduling and execution) by executing SDF graphs using multiple software threads. Our target applications are modern wireless communication and signal processing systems. According to Chapter 7, SDF representations of such systems typically result in *critical* SDF graphs that challenge simulations — here, by critical, we mean an SDF graph that has large-scale (a large number of actors and edges), complex topology, and heavily multirate behavior.

The key problem behind multithreaded SDF simulation is scheduling SDF graphs for on-chip, thread-level parallel computation. Scheduling in our context consists of the following related tasks:

1. *Clustering* — Partitioning and clustering actors in the SDF graph into multiple clusters such that actors in the same cluster are executed sequentially by a single software thread. This task is analogous to "assignment" in multiprocessor scheduling.
2. *Ordering* — Ordering multiple firings of the same actor as well as firings across different actors inside each cluster. This task is similar to "ordering" in multiprocessor scheduling, but involves additional considerations for satisfying

multirate SDF consistency.

3. *Buffering* — Computing buffer sizes for edges inside and across clusters. In dataflow semantics, edges generally represent infinite FIFO buffers, but for practical implementations, it is necessary to impose such bounds on buffer sizes.
4. *Assignment* — Creating certain numbers of threads, and assigning clusters to threads for concurrent execution, under the constraint that each cluster can only be executed by one software thread at any given time.
5. *Synchronization* — Determining when a cluster is executed by a software thread, and synchronizing between multiple concurrent threads such that all data precedence and buffer bound constraints are satisfied. This task is analogous to "timing" in multiprocessor scheduling.

Scheduling SDF graphs for multithreaded simulation is quite different than scheduling HSDF graphs for embedded multiprocessor implementation. In our context, software threads present additional exploration space between SDF graphs and processing units. Creation and usage of software threads are part of scheduling tasks (assignment); while the operating system schedules the usage of processing units among threads that come from the simulation process as well as from other processes. In addition, because our objective is to speed up simulation runtime, low-complexity scheduling is of major concern; while for embedded multiprocessor implementations, tolerance for compile-time is relatively high (e.g., in multiprocessor scheduling, an SDF graph is often converted to an equivalent HSDF graph [8], and this can in general exponentially increase the number of actors). Furthermore,

our focus is on *long term* simulation — for satisfactory simulation, SDF graphs are executed over and over again for significant numbers of iterations. As a result, to speed up simulation runtime, throughput is one of the key factors; while latency has relatively low priority.

In this thesis, we develop the *multithreaded simulation scheduler* (MSS) to systematically exploit multithreading capabilities when simulating SDF-based designs. The compile-time scheduling framework in MSS strategically integrates graph clustering, actor vectorization, intra-cluster scheduling, and inter-cluster buffering techniques to jointly perform static clustering, static ordering, and static buffering for trading off between throughput, synchronization overhead, and buffer requirements. From this compile-time framework, *inter-thread communication* (ITC) SDF graphs are constructed for multithreaded execution. The runtime scheduling in MSS then applies either the self-timed (static assignment) or self-scheduled (dynamic assignment) multithreaded execution model to schedule and synchronize multiple software threads for executing ITC graphs at runtime.

The organization of this chapter is as follows: We review related background in Section 8.2. In Section 8.3, we present Ω-scheduling, the theoretical foundation of MSS. We then introduce our compile-time scheduling framework in Section 8.4, and our runtime scheduling approach in Section 8.5. In Section 8.6, we demonstrate simulation results, and we conclude in the final section.

8.2 Background

Synchronous dataflow (SDF) and SDF scheduling preliminaries are presented in Section 2.1, and SDF clustering is discussed in Section 7.3.1. Homogeneous synchronous dataflow (HSDF) is introduced in Section 2.2. HSDF is widely used in throughput analysis and multiprocessor scheduling. Any consistent SDF graph can be converted to an equivalent HSDF graph based on the *SDF-to-HSDF transformation* [75] such that samples produced and consumed by every invocation of each actor in the HSDF graph remain identical to those in the original SDF graph.

Let Z^+ denote the set of positive integers. Given an HSDF graph $G = (V, E)$, we denote the execution time of an actor v by $t(v)$, and denote $f_T : V \rightarrow Z^+$ as an actor execution time function that assigns $t(v)$ to a finite positive integer for every $v \in V$ (the actual execution time $t(v)$ can be interpreted as cycles of a base clock).

The *cycle mean* (CM) of a cycle c in an HSDF graph is defined as

$$CM(c) = \frac{\sum_{v \text{ in } c} t(v)}{\sum_{e \text{ in } c} del(e)}. \tag{8.1}$$

The *maximum cycle mean* (MCM) of an HSDF graph G is defined as

$$MCM(G) = \max_{\text{cycle } c \text{ in } G} CM(c). \tag{8.2}$$

Theorem 8.1. *[68] Given a strongly connected HSDF graph G, when actors execute as soon as data is available at all inputs, the iteration period is $MCM(G)$, and the maximum achievable throughput is $1/MCM(G)$.*

8.3 Ω-Scheduling

As described in Section 8.1, the problem of scheduling SDF graphs for multithreaded execution is highly complex. Our first step is to assume unbounded processing resources — that is, we can schedule the graph with as many processing resources as desired. In this case, the scheduling tasks of clustering, ordering, and assignment become trivial because the best strategy is to assign each actor exclusively to a processor. Then the problem can be simplified as follows: given unbounded processing resources, how do we schedule (including buffering and synchronization) SDF graphs to achieve maximal throughput? In this section, we develop a set of theorems and algorithms to solve this problem. Note that the developments in this section are not specific to multithreaded execution, and we envision that they can be applied to many contexts for parallel execution of SDF graphs.

8.3.1 Definitions and Methods for Throughput Analysis

We assume a graph starts execution at time $t = 0$. We denote the count of complete firings of an actor v until time t since the graph starts execution by $ct(v, t)$. Note that by definition, $ct(v, 0) = 0$. We denote the number of tokens — the *state* — on an edge e at time t by

$$\begin{aligned} tok(e,t) = & ct(src(e),t) \times prd(e) - \\ & ct(snk(e),t) \times cns(e) + del(e). \end{aligned} \tag{8.3}$$

We assume actor firing is an *atomic* operation, and define that the *state transition* — i.e., the change in the number of tokens on an edge e happens imme-

diately when either $src(e)$ or $snk(e)$ finishes its firing (execution). Suppose actor v starts execution at time t_0, and suppose none of v's adjacent actors finish execution between t_0 and $t_1 = t_0 + t(v)$. Then at time t_1, v finishes its execution, and for every $e \in in(v)$, $tok(e, t_1) = tok(e, t_0) - cns(e)$, and for every $e \in out(v)$, $tok(e, t_1) = tok(e, t_0) + prd(e)$.

In dataflow-related tools, actors may have internal state that prevents executing multiple invocations of the actors in parallel, e.g., FIR filters. Furthermore, whether or not an actor has internal state may be a lower level detail in the actor's implementation that is not visible to the tool (e.g., to algorithms that operate on the application dataflow graph). This is, for example, the case in Agilent ADS, the specific design tool that provides the context for our study and the platform for our experiments. Thus, exploring data-level parallelism by duplicating actors onto multiprocessors, e.g., [71], is out of the scope of this thesis.

In pure dataflow semantics, data-driven execution simply assumes infinite edge buffers. For practical implementations, it is necessary to impose bounds on buffer sizes. Given an SDF graph $G = (V, E)$, we denote the buffer size of an edge $e \in E$ (i.e., the bound on the size of a FIFO buffer or the size of a circular buffer [4] or other types of buffer implementations) by $buf(e)$, and denote $f_B : E \rightarrow Z^+$ as a buffer size function that assigns $buf(e)$ to a finite positive integer for every $e \in E$. In the following definition, we refine the fireable condition to take bounded-buffers into account.

Definition 8.2. Given an SDF graph $G = (V, E)$ and a buffer size function f_B, an

actor $v \in V$ is (*data-driven*) *bounded-buffer fireable* at time t if 1) v is fireable — i.e., v has sufficient numbers of tokens on all of its input edges (data-driven property) — $\forall e \in in(v)$, $tok(e,t) \geq cns(e)$, and 2) v has sufficient numbers of spaces on all of its output edges (bounded-buffer property) — $\forall e \in out(v)$, $buf(e) - tok(e,t) \geq prd(e)$.

In the rest of the chapter, we use the term "fireable" to indicate conventional data-driven semantics, and the term "data-driven bounded-buffer fireable" or simply "bounded-buffer fireable" to indicate the data-driven semantics implied by Definition 8.2.

Recall that the task of synchronization is to maintain data precedence and bounded-buffer constraints. As a result, the most intuitive scheduling strategy for maximal throughput is to fire an actor as soon as it is bounded-buffer fireable. We define such a scheduling strategy as follows, where actors are synchronized by bounded-buffer fireability.

Definition 8.3. Given a consistent SDF graph $G = (V, E)$, Ω-*scheduling* is defined as the SDF scheduling strategy that 1) statically assigns each actor $v \in V$ to a separate processing unit, 2) statically determines a buffer bound $buf(e)$ for each edge $e \in E$, and 3) fires an actor as soon as it is bounded-buffer fireable.

In the following definitions, we define the concepts of the Ω-SDF graph and the Ω-HSDF graph, which are important to throughput analysis for Ω-scheduling.

Definition 8.4. Given an SDF graph $G = (V, E)$ and a buffer size function f_B, the Ω-*SDF graph* of G is defined as $G^{\Omega} = (V, \{E + E_b^{\Omega} + E_s^{\Omega})\}$. Here, E_s^{Ω} is the set of self-loops that models the sequential execution constraint for multiple fir-

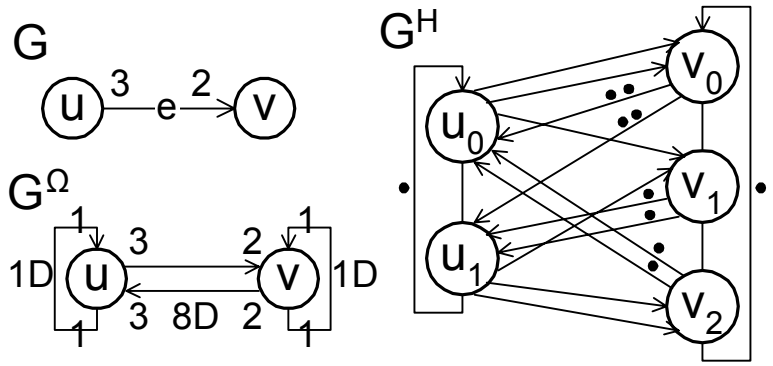


Figure 8.1: SDF graph G, Ω-SDF graph G^{Ω} given $buf(e) = 8$, and the transformed Ω-HSDF graph G^H.

ings of the same actor (on a single processing unit) — that is, for each actor $v \in V$, there is an edge $e_s \in E_s^{\Omega}$ such that $src(e_s) = v$, $snk(e_s) = v$, $prd(e_s) = 1$, $cns(e_s) = 1$, $del(e_s) = 1$. E_b^{Ω} is the set of edges that models the bounded-buffer constraint in Ω-scheduling — that is, for each edge $e \in E$, there is a corresponding edge $e_b \in E_b^{\Omega}$ such that $src(e_b) = snk(e)$, $snk(e_b) = src(e)$, $prd(e_b) = cns(e)$, $cns(e_b) = prd(e)$, $del(e_b) = buf(e) - del(e)$.

Definition 8.5. Given an SDF graph $G = (V, E)$ and a buffer size function f_B, the *Ω-HSDF graph* G^H of G is defined as the HSDF graph that is transformed from the Ω-SDF graph G^{Ω} based on the SDF-to-HSDF transformation (as described in Section 8.2).

Figure 8.1 presents an SDF graph G, the Ω-SDF graph G^{Ω} given $buf(e) = 8$, and the corresponding Ω-HSDF graph G^H. Next, we analyze the throughput upper bound for Ω-scheduling in the following theorems.

Theorem 8.6. *Suppose that we are given an SDF graph $G = (V, E)$ and a buffer size function f_B. An actor is bounded-buffer fireable in G if and only if it is fireable*

in the Ω-SDF graph G^{Ω}.

Proof. By Definition 8.4 and Equation (8.4), we can derive that for each $e \in E$ and its bounded-buffer counterpart $e_b \in E_b^{\Omega}$, the data-driven condition $tok(e_b, t) \geq cns(e_b)$ in G^{Ω} is equivalent to the bounded-buffer condition $buf(e) - tok(e, t) \geq prd(e)$ in G.

$$\begin{aligned} tok(e_b, t) =& ct(src(e_b), t) \times prd(e_b) - \\ & ct(snk(e_b), t) \times cns(e_b) + del(e_b) \\ =& ct(snk(e), t) \times cns(e) - \\ & ct(src(e), t) \times prd(e) + buf(e) - del(e) \\ =& buf(e) - tok(e, t) \end{aligned} \tag{8.4}$$

Then by Definition 8.2 and the fact that E_s^{Ω} does not affect the fireable condition, the proof is complete. □

Theorem 8.7. *Suppose that we are given a consistent SDF graph G, a buffer size function f_B, and an actor execution time function f_T. Then the maximum achievable throughput in Ω-scheduling is the inverse of the maximum cycle mean of the corresponding Ω-HSDF graph G^H —*

$$\frac{1}{MCM(G^H)}. \tag{8.5}$$

Proof. By Definition 8.4, the Ω-SDF graph G^{Ω} is strongly connected. Then based on the SDF-to-HSDF transformation (as described in Section 8.2), G^{Ω} and G^H are equivalent, and G^H is strongly connected. Finally by Theorem 8.6 and Theorem 8.1, the proof is complete. □

Theorem 8.8. *Suppose that we are given a consistent, acyclic SDF graph $G = (V, E)$ and an actor execution time function f_T. Then the maximum achievable throughput in Ω-scheduling is*

$$\frac{1}{\max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))}. \tag{8.6}$$

Proof. Suppose for every edge $e \in E$, $buf(e)$ is assigned a positive integer that approaches infinity ∞. Let $G^H = (V^H, \{E^H + E_b^H + E_s^H)\}$ denote the Ω-HSDF graph transformed from the Ω-SDF graph $G^\Omega = (V, \{E + E_b^\Omega + E_s^\Omega)\}$, where V^H, E^H, E_b^H, and E_s^H are transformed from V, E, E_b^Ω, and E_s^Ω, respectively. Let C denote the set of all cycles in G^H, and let C_s denote the set of cycles transformed from self-loops E_s^Ω. Based on the SDF-to-HSDF transformation (as described in Section 8.2), we can derive that: 1) for every $e_b \in E_b^H$, $del(e_b) \to \infty$; 2) for every $c \in \{C - C_s\}$, c contains at least one edge $e_b \in E_b^H$, and as a result, $CM(c) \to 0$; and 3) for every $v \in V$, there exists a cycle $c \in C_s$ transformed from the corresponding self-loop $e_s \in E_s^\Omega$, and thus, $CM(c) = \boldsymbol{q}_G[v] \times t(v)$. Therefore,

$$MCM(G^H) = \max_{c \in C_s} CM(c) = \max_{v \in V}(\boldsymbol{q}_G[v] \times t(v)).$$

By Theorem 8.7, the proof is complete. □

Theorem 8.9. *Given a consistent, acyclic SDF graph $G = (V, E)$ and an actor execution time function f_T, there exists a finite buffer size function f_B that gives the maximum achievable throughput — Equation (8.6) — in Ω-scheduling.*

Proof. This result involves the same hypotheses as Theorem 8.8, and we prove this result by continuing with the notations and arguments given in the proof of Theorem

8.8. First, we define $T = \max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$ and $L = \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$. Now suppose f_B sets

$$buf(e) = T \times L \times prd(e) \times \boldsymbol{q}_G[src(e)] + del(e)$$

for each edge $e \in E$. Based on the SDF-to-HSDF transformation (as described in Section 8.2), we can derive that: 1) for every $e_b \in E_b^H$, $del(e_b) \geq T \times L$; 2) for every $c \in \{C - C_s\}$, c contains at least one edge $e_b \in E_b^H$, and because L is an upper bound on the cycle length in G^H, $CM(c) \leq T$; and 3) for every $v \in V$, there exists a cycle $c \in C_s$ transformed from the corresponding self-loop $e_s \in E_s^\Omega$, and again, $CM(c) = \boldsymbol{q}_G[v] \times t(v)$. Based on the above derivation and by Theorem 8.7, the maximum achievable throughput is

$$1/MCM(G^H) = 1/\max_{c \in C} CM(c) = 1/T.$$

Finally, because for every $v \in V$, $\boldsymbol{q}_G[v]$ and $t(v)$ are finite, we have that $buf(e)$ is finite for every $e \in E$. □

Theorem 8.10. *Given a consistent SDF graph $G = (V, E)$ and an actor execution time function f_T, Equation* (8.6) *is the throughput upper bound in* Ω*-scheduling — that is, the maximum achievable throughput is less than or equal to Equation* (8.6).

Proof. Let $G^H = (V^H, \{E^H + E_b^H + E_s^H)\}$ denote the Ω-HSDF graph transformed from the Ω-SDF graph $G^\Omega = (V, \{E + E_b^\Omega + E_s^\Omega)\}$, where V^H, E^H, E_b^H, and E_s^H are transformed from V, E, E_b^Ω, and E_s^Ω, respectively. Let C denote the set of all cycles in G^H; let C_s denote the set of cycles transformed from self-loops E_s^Ω; let C_b denote the set of cycles that contain at least one $e_b \in E_b^H$; and let $C_c = C - C_s - C_b$.

Suppose that G contains one or more cycles. Then based on the SDF-to-HSDF transformation (as described in Section 8.2), we can derive that $C_c \neq \varnothing$ and each cycle in C_c is transformed from a corresponding cycle in G. Given sufficient buffer sizes such that C_b does not dominate the maximum cycle mean (as with the proof of Theorem 8.9), we have that

$$MCM(G^H) = \max_{c \in \{C_s + C_c\}} CM(c) \geq \max_{c \in C_s} CM(c) = \max_{v \in V} (\boldsymbol{q}_G[v] \times t(v)).$$

By Theorem 8.7 and Theorem 8.8, the proof is complete. □

In summary, Equation (8.6) is an upper bound on throughput for a consistent SDF graph in Ω-scheduling, and is the maximum achievable throughput for a consistent, acyclic SDF graph in Ω-scheduling.

8.3.2 Buffering

Based on the previous derivations, providing sufficient buffer sizes is important to achieve maximum achievable throughput in Ω-scheduling. In this subsection, we develop buffering techniques for Ω-scheduling. We start with a few definitions.

A dataflow graph $G = (V, E)$ is in general a directed *multigraph* (i.e., multiple edges can have the same source and sink vertices). Here, we define a *parallel edge set* $[u, v]$ as a set of edges $\{e \in E | src(e) = u \text{ and } snk(e) = v\}$ that connect from the same source vertex u to the same sink vertex v. The following property, definition, and theorem are useful in abstracting parallel edge sets in order to simplify the developments of buffering techniques.

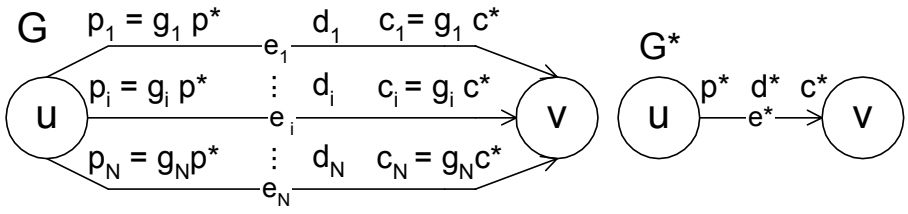


Figure 8.2: (a) Consistent parallel edge set. (b) Primitive edge.

Property 8.11. *Given a consistent SDF graph $G = (V, E)$. Every parallel edge set $[u, v] \in E$ has a general form as shown in Figure 8.2(a), where for each edge $e_i \in [u, v]$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, and $c^* = c_i/g_i$. For consistency, the coprime positive integers p^* and c^* must satisfy $p_i/c_i = p^*/c^*$ for every $e_i \in [u, v]$.*

Definition 8.12. Given a consistent SDF graph $G = (V, E)$, the *primitive graph* $G^* = (V, E^*)$ is constructed by replacing each parallel edge set $[u, v] \in E$, as shown in Figure 8.2(a), with a single edge e^*, as shown in Figure 8.2(b), where $prd(e^*) = p^*$, $cns(e^*) = c^*$, $del(e^*) = d^* = \min_{e_i \in [u,v]} \lfloor d_i/g_i \rfloor$, and $\gcd(p^*, c^*) = 1$. The edge e^* is called the *primitive edge* of $[u, v]$, and the values p^*, c^*, and d^* are called the *primitive production rate*, *primitive consumption rate*, and *primitive delay* of $[u, v]$, respectively. An edge $e_i \in [u, v]$ that satisfies $\lfloor d_i/g_i \rfloor = d^*$ is called a *maximally-constrained edge* of $[u, v]$.

Property 8.11 and Definition 8.12 are the generalized versions of two-actor SDF graphs defined in Section 7.3.9.

Theorem 8.13. *Given a consistent SDF graph $G = (V, E)$ and its primitive graph $G^* = (V, E^*)$, and given the same actor execution function f_T for both G and G^*,*

suppose Equation (8.7) *is satisfied for each parallel edge set* $[u, v] \in E$ *and its primitive edge* $e^* \in E^*$.

$$\forall e_i \in [u, v], buf(e_i) = buf(e^*) \times g_i + d_i - d^* \times g_i \tag{8.7}$$

Then Ω*-scheduling for* G *is equivalent to* Ω*-scheduling for* G^* *— that is, an actor is bounded-buffer fireable in* G *if and only if it is bounded-buffer fireable in* G^*.

Proof. Initially at time $t = 0$, for each $v \in V$, $ct(v, 0) = 0$ for both G and G^*. Suppose that for some time $t \geq 0$, for each $v \in V$, $ct(v, t)$ is the same for both G and G^*. Then at this time t, we can derive that Equation (8.8) and Equation (8.9) are satisfied for every parallel edge set $[u, v] \in E$ and its primitive edge $e^* \in E^*$.

$$\begin{aligned}
& \forall e_i \in [u, v], tok(e_i, t) \geq c_i \\
\Leftrightarrow\ & \forall e_i \in [u, v], ct(u, t) \times p_i - ct(v, t) \times c_i + d_i \geq c_i \\
\Leftrightarrow\ & \forall e_i \in [u, v], ct(u, t) \times p^* \times g_i - ct(v, t) \times c^* \times g_i + \\
& \lfloor d_i/g_i \rfloor \times g_i + d_i \bmod g_i \geq c^* \times g_i \\
\Leftrightarrow\ & \forall e_i \in [u, v], ct(u, t) \times p^* - ct(v, t) \times c^* + \lfloor d_i/g_i \rfloor \geq c^* \\
\Leftrightarrow\ & ct(u, t) \times p^* - ct(v, t) \times c^* + d^* \geq c^* \\
\Leftrightarrow\ & tok(e^*, t) \geq c^*
\end{aligned} \tag{8.8}$$

$$
\begin{aligned}
& \forall e_i \in [u,v], buf(e_i) - tok(e_i,t) \geq p_i \\
\Leftrightarrow\ & \forall e_i \in [u,v], buf(e_i) - (ct(u,t) \times p_i - ct(v,t) \times c_i + d_i) \geq p_i \\
\Leftrightarrow\ & \forall e_i \in [u,v], buf(e^*) \times g_i + d_i - d^* \times g_i - \\
& (ct(u,t) \times p^* \times g_i - ct(v,t) \times c^* \times g_i + d_i) \geq p^* \times g_i \\
\Leftrightarrow\ & buf(e^*) - (ct(u,t) \times p^* - ct(v,t) \times c^* + d^*) \geq p^* \\
\Leftrightarrow\ & buf(e^*) - tok(e^*,t) \geq p^*
\end{aligned}
\tag{8.9}
$$

Thus, at time t, for each $v \in V$, v is bounded-buffer fireable in G if and only if it is bounded-buffer fireable in G^*. Because Ω-scheduling fires an actor as soon as it is bounded-buffer fireable, we have that at time $(t+1)$, for each $v \in V$, $ct(v, t+1)$ is the same for both G and G^*. By mathematical induction, the proof is complete. □

The following property observes the periodic behavior in Ω-scheduling, and is important to our developments.

Property 8.14. *Given a consistent SDF graph $G = (V, E)$, a buffer size function f_B, and an actor execution time function f_T, suppose the iteration period of G under Ω-scheduling is T. Then after a finite* transient *phase, each actor $v \in V$ can enter a* periodic *phase such that v fires $\boldsymbol{q}_G[v]$ times in period T. Furthermore, for any kth firing of v in the periodic phase, the time between the kth and the $(k + \boldsymbol{q}_G[v])$th firing is T.*

Note that the time for each actor to enter its periodic phase may be different. In other words, the periods for different actors may not align to the same time instances.

Many existing techniques for joint buffer and throughput analyses rely on prior knowledge of actor execution times. However, exact actor execution time information may be unavailable in practical situations. In this thesis, we focus on minimizing buffer requirements under the maximum achievable throughput in Ω-scheduling without prior knowledge of actor execution time. In the following theorem, we first provide such analysis for two-actor SDF graphs.

Theorem 8.15. *Given a consistent, acyclic, two-actor SDF graph $G = (\{u, v\}, [u, v])$, the minimum buffer size to sustain the maximum achievable throughput in Ω-scheduling over any actor execution time function is given by Equation* (8.10)*:*

$$\forall e_i \in [u,v], buf(e_i) = \begin{cases} (p_i + c_i - g_i) \times 2 + d_i - d^* \times g_i \ , \\ \quad \text{if } 0 \leq d^* \leq (p^* + c^* - 1) \times 2 \\ d_i, \ \text{otherwise} \end{cases} \tag{8.10}$$

Here, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, $c^* = c_i/g_i$, and $d^* = \min_{e_i \in [u,v]} \lfloor d_i/g_i \rfloor$.

Proof. Without loss of generality, suppose that G is in the general form shown in Figure 8.2(a), and suppose Figure 8.2(b) represents the primitive graph G^*. Based on Theorem 8.13, we first solve the problem for the primitive graph G^* and then apply Equation (8.7).

Suppose $c^* \times t(u) = p^* \times t(v) = T$. Based on Theorem 8.8 and Property 8.14, in order to achieve the maximum achievable throughput in the periodic phase, u must execute c^* times continuously, and v must execute p^* times continuously

in a period T. Let t_0 denote the time when u and v enter the periodic phase, and denote $tok(e^*, t_0)$ by n_0. By state enumeration of the ith period in the periodic phase ($i \in \{0, 1, \dots\}$), there is a total of $p^* + c^* - 1$ different states in a period T — that is, 1) $tok(e^*, t_0 + i \times T) = n_0$, 2) for each $k \in \{1, 2, \dots, p^* - 1\}$, $tok(e^*, t_0 + i \times T + k \times t(v))$ is a unique state from $\{n_0 - (p^* - 1), n_0 - (p^* - 2), \dots, n_0 - 1\}$, and 3) for each $k \in \{1, 2, \dots, c^* - 1\}$, $tok(e^*, t_0 + i \times T + k \times t(u))$ is a unique state from $\{n_0 + 1, n_0 + 2, \dots, n_0 + c^* - 1\}$. Figure 8.3 illustrates the state enumeration for $p^* = 5$, $c^* = 3$, $i = 0$, and $t(u) \times 3 = t(v) \times 5$.

For each $k \in \{1, 2, \dots, p^* - 1\}$, in order for v to be fireable immediately at time $t_0 + i \times T + k \times t(v)$, we must have $tok(e^*, t_0 + i \times T + k \times t(v)) \geq c^*$. As a result, $n_0 \geq (p^* - 1) + c^*$ must be satisfied to achieve the maximum achievable throughput. On the other hand, for each $k \in \{1, 2, \dots, c^* - 1\}$, in order for u to be fireable immediately at time $t_0 + i \times T + k \times t(u)$, we must have $buf(e^*) - tok(e^*, t_0 + i \times T + k \times t(u)) \geq p^*$. Thus, to achieve the maximum achievable throughput, $buf(e^*) \geq n_0 + c^* - 1 + p^*$ must be satisfied.

Now consider the case where $d^* \leq (p^* + c^* - 1) \times 2$. Because p^* and c^* are co-prime, $tok(e^*, t_0) = n_0 = p^* + c^* - 1$ is a reachable state from $tok(e^*, 0) = d^*$ in the transient phase. Therefore, $buf(e^*) = (p^* + c^* - 1) \times 2$ is the minimum buffer size for G^* to achieve the maximum achievable throughput in Ω-scheduling, — i.e., $1/\max(c^* \times t(u), p^* \times t(v))$ by Equation (8.6), or equivalently,

$$MCM(G^{*H}) = CM(c_u) = c^* \times t(u) = CM(c_v) = p^* \times t(v),$$

where G^{*H} denotes the Ω-HSDF graph of G^*, and c_u and c_v denote the cycles in G^{*H} transformed from the self-loops of u and v, respectively.

On the other hand, when $d^* > (p^* + c^* - 1) \times 2$, $buf(e^*) = d^*$ is the minimum buffer size to achieve the maximum achievable throughput because $buf(e^*)$ must accommodate d^* initial tokens, and larger d^* may only decrease cycle means resulting from e^* and its bounded-buffer counterpart, but does not affect $CM(c_u)$ nor $CM(c_v)$.

In summary, when $c^* \times t(u) = p^* \times t(v) = T$, Equation (8.11) can achieve the maximum achievable throughput $1/\max(c^* \times t(u), p^* \times t(v)) = 1/T$.

$$buf(e^*) = \begin{cases} (p^* + c^* - 1) \times 2, \text{if } 0 \leq d^* \leq (p^* + c^* - 1) \times 2 \\ d^*, \text{ otherwise} \end{cases} \tag{8.11}$$

When $c^* \times t(u) > p^* \times t(v)$, by applying Equation (8.11) on G^*, we can derive that smaller $t(v)$ reduces $CM(c_v)$ as well as other cycle means that involve v, but does not affect $CM(c_u)$. As a result,

$$MCM(G^{*H}) = CM(c_u) = \max(c^* \times t(u), p^* \times t(v)).$$

On the other hand, when $c^* \times t(u) < p^* \times t(v)$, by applying Equation (8.11) on G^*, we can derive in a similar manner that

$$MCM(G^{*H}) = CM(c_v) = \max(c^* \times t(u), p^* \times t(v)).$$

Therefore, Equation (8.11) can sustain the maximum achievable throughput regardless of $t(u)$ and $t(v)$. Finally, by substituting Equation (8.11) into Equation (8.7), the proof is complete. □

Note that given exact actor execution times, the minimum buffer requirement to achieve the maximum achievable throughput in Ω-scheduling may be less than

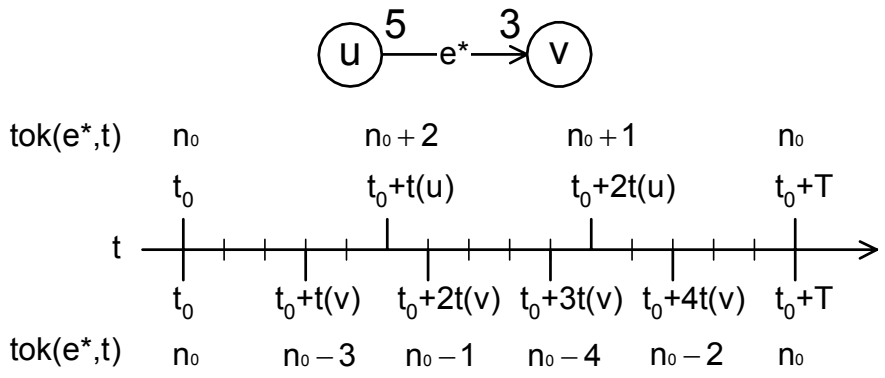


Figure 8.3: State enumeration for $p^* = 5$, $c^* = 3$, and $t(u) \times 3 = t(v) \times 5$.

Equation (8.10). However, Equation (8.10) is the minimum buffer requirement to "sustain" the maximum achievable throughput in Ω-scheduling over arbitrary actor execution times. In order to generalize the above derivation to acyclic SDF graphs, it is useful to employ the notion of biconnected components.

Definition 8.16. Given a connected graph $G = (V, E)$, a *biconnected component* is a maximal set of parallel edge sets $A \subseteq E$ such that any pair of parallel edge sets in A lies in a simple undirected cycle. A *bridge* is then a parallel edge set that does not belong to any biconnected component, or equivalently, a parallel edge set whose removal disconnects G.

Traditionally, biconnected components and bridges are defined with respect to single edges [16] rather than parallel edge sets. Because we are only interested in the topology formed by parallel edge sets, we adapt the original definition to our context. For example, Figure 8.4 shows biconnected components and bridges of an example graph, where bridges are marked with dashed lines.

In the following theorem, we generalize Theorem 8.15 to acyclic SDF graphs

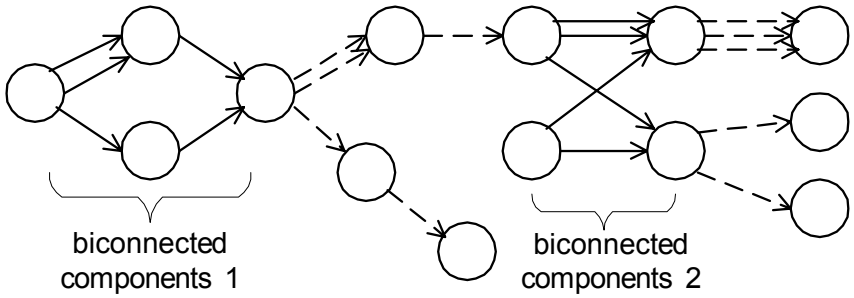


Figure 8.4: Biconnected components and bridges.

that do not contain biconnected components.

Theorem 8.17. *Suppose that we are given a consistent SDF graph $G = (V, E)$, and suppose that G does not contain any biconnected components. Then the minimum buffer sizes to sustain the maximum achievable throughput in Ω-scheduling over any actor execution time function are given by Equation* (8.12)*:*

$$\forall [u, v] \in E$$

$$\forall e_i \in [u, v], buf(e_i) = \begin{cases} (p_i + c_i - g_i) \times 2 + d_i - d^* \times g_i \ , \\ \quad \text{if } 0 \leq d^* \leq (p^* + c^* - 1) \times 2 \\ d_i, \ \text{otherwise} \end{cases} \tag{8.12}$$

Here, for each parallel edge set $[u, v] \in E$, and for each edge $e_i \in [u, v]$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, $c^* = c_i/g_i$, and $d^* = \min_{e_i \in [u,v]} \lfloor d_i/g_i \rfloor$.

Proof. Without loss of generality, let $L = v_1, v_2, \ldots, v_{|V|}$ denote a traversal of G such that any actor v_k in L is adjacent to at least one actor in $\{v_1, v_2, \ldots, v_{k-1}\}$. For example, L can be constructed from any undirected depth-first or breadth-first traversal. Let $V_{1,k} = \{v_1, v_2, \ldots, v_k\}$ denote a subset of the first k actors in L. Let

$G_{1,k} = (V_{1,k}, E_{1,k})$ denote a subgraph consisting of $V_{1,k}$ and $E_{1,k} = \{e \in E | src(e) \in V_{1,k} \text{ and } snk(e) \in V_{1,k}\}$. Lastly, let us denote $\gcd_{v \in V_{1,k}} \boldsymbol{q}_G[v]$ by $g_{V_{1,k}}$.

Suppose for every $v \in V$, $\boldsymbol{q}_G[v] \times t(v)$ equals a constant value T. Based on Theorem 8.8 (here G is acyclic because it contains no biconnected components) and Property 8.14, in order to achieve the maximum achievable throughput in the periodic phase, every actor $v \in V$ must fire continuously at rate $\boldsymbol{q}_G[v]/T$ after a finite transient phase.

Initially for $G_{1,2}$, by Theorem 8.15, given Equation (8.10) in Ω-scheduling, v_1 can fire continuously at rate $(\boldsymbol{q}_G[v_1]/g_{V_{1,2}})/(T/g_{V_{1,2}})$ after a finite transient phase, and v_2 can fire continuously at rate $(\boldsymbol{q}_G[v_2]/g_{V_{1,2}})/(T/g_{V_{1,2}})$ after a finite transient phase. Next, for a particular $G_{1,k}$, where $2 < k < |V|$, suppose given Equation (8.12) in Ω-scheduling, each actor $v \in V_{1,k}$ can fire continuously at rate $(\boldsymbol{q}_G[v]/g_{V_{1,k}})/(T/g_{V_{1,k}})$ after a finite transient phase.

Now consider $G_{1,k+1}$. Because G has no biconnected components, there is one and only one actor $u \in V_{1,k}$ that connects v_{k+1}. By Theorem 8.15, given Equation (8.10) for the two-actor graph $[u, v_{k+1}]$ (or $[v_{k+1}, u]$) alone in Ω-scheduling, u can fire continuously at rate $(\boldsymbol{q}_G[u]/g_{\{u,v_{k+1}\}})/(T/g_{\{u,v_{k+1}\}})$ after a finite transient phase, and v_{k+1} can fire continuously at rate $(\boldsymbol{q}_G[v_{k+1}]/g_{\{u,v_{k+1}\}})/(T/g_{\{u,v_{k+1}\}})$ after a finite transient phase, where $g_{\{u,v_{k+1}\}} = \gcd(\boldsymbol{q}_G[u], \boldsymbol{q}_G[v_{k+1}])$.

Based on the observations above, we can derive that given Equation (8.12) for $G_{1,k+1}$, Ω-scheduling is able to make each actor $v \in V_{1,k+1}$ fire continuously at rate $(\boldsymbol{q}_G[v]/g_{V_{1,k+1}})/(T/g_{V_{1,k+1}})$ after a finite transient phase — this is because 1)

Ω-scheduling can always delay an actor firing until it is bounded-buffer fireable for adapting to the effect of inserting v_{k+1}, and 2) the time to enter the periodic phases of $G_{1,k}$ and $[u, v_{k+1}]$ (or $[v_{k+1}, u]$) is finite.

By mathematical induction, given Equation (8.12) for G in Ω-scheduling, each actor $v \in V$ can fire continuously at rate $\boldsymbol{q}_G[v]/T = 1/t(v)$ after a finite transient phase. As a result, by Property 8.14, the iteration period of G in Ω scheduling is $MCM(G^H) = T$, where G^H is the Ω-HSDF graph of G. Furthermore, by Theorem 8.15, Equation (8.10) is the minimum buffer sizes for each acyclic pair of adjacent actors to fire continuously in the periodic phase. Therefore, Equation (8.12) gives the minimum buffer sizes to achieve the maximum achievable throughput, or equivalently, minimum achievable iteration period $MCM(G^H) = T$ in Ω-scheduling when $\boldsymbol{q}_G[v] \times t(v) = T$ for every $v \in V$.

Now consider the case where $\boldsymbol{q}_G[v] \times t(v)$ is not constant across $v \in V$. Let us denote $\max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$ by T, and denote $V_{max} = \{v \in V | \boldsymbol{q}_G[v] \times t(v) = T\}$. Compared to the above case where where $\boldsymbol{q}_G[v] \times t(v) = T$ for every $v \in V$, for an actor $v \in \{V - V_{max}\}$, the smaller $\boldsymbol{q}_G[v] \times t(v)$ only reduces cycle means involving v, but does not affect the cycle means of the self loops of V_{max}. As a result, given Equation (8.12) for G in Ω-scheduling, $MCM(G^H) = \max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$.

Based on the above derivations, Equation (8.12) gives the minimum buffer sizes to sustain the maximum achievable throughput in Ω-scheduling over any actor execution time function. □

Applying Equation (8.12) to general acyclic SDF graphs may cause deadlock

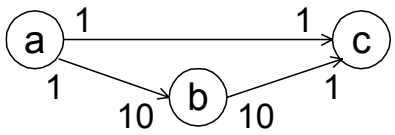


Figure 8.5: Buffering deadlock example.

Ω-ACYCLIC-BUFFERING(G)
input: a consistent acyclic SDF graph $G = (V, E)$
1 E_B = BRIDGES(G)
2 **for** each $[u, v] \in E_B$ compute buffer sizes by Equation (8.10) **end**
3 $\{E_1, E_2, \ldots, E_N\}$ = BICONNECTED-COMPONENTS(G)
4 **for** each biconnected subgraph $G_i = (V_i, E_i)$ from $i = 1$ to N
5 $\{V_i^1, V_i^2, \ldots, V_i^M\}$ = BICONNECTED-FREE-PARTITION(G_i)
6 $G_i' = (V_i', E_i')$ = CLUSTER(G_i, $\{V_i^1, V_i^2, \ldots, V_i^M\}$)
7 compute buffer sizes for E_i' by Equation (8.12) on G_i'
8 **for** each partitioned subgraph $G_i^j = (V_i^j, E_i^j)$ from $j = 1$ to M
9 Ω-ACYCLIC-BUFFERING(G_i^j)
10 **end**
11 **end**

Figure 8.6: Ω-Acyclic-Buffering algorithm.

in Ω-scheduling. Figure 8.5 presents such an example: if the buffer size for edge (a, c) is set to 2, then the graph is deadlocked because neither b nor c can fire due to insufficient buffer size on (a, c). In order to allocate buffers for general acyclic SDF graphs in Ω-scheduling, we have developed the Ω-*Acyclic-Buffering* algorithm as shown in Figure 8.6, and we prove the validity of the algorithm in Theorem 8.18.

In Figure 8.6, we compute bridges E_B of G and set buffer sizes of each parallel edge set in E_B by Equation (8.10) in lines 1-2. Next, we compute the biconnected components $E_1, E_2, \ldots, E_N$ of G in line 3. For each biconnected subgraph $G_i = (V_i, E_i)$ induced from the biconnected component E_i (where V_i is the set of source and sink actors of edges in E_i) for $i \in \{1, 2, \ldots, N\}$, we first compute a *biconnected-free partition* $V_i^1, V_i^2, \ldots, V_i^M$ of V_i in line 5 such that clustering $V_i^1, V_i^2, \ldots, V_i^M$ in G_i in line 6 does not introduce any biconnected component in

the clustered version $G'_i = (V'_i, E'_i)$ of G_i. After that, we apply Equation (8.12) on G'_i to compute buffer sizes for E'_i. Then in line 8, we apply the Ω-Acyclic-Buffering algorithm *recursively* to each acyclic subgraph $G_i^j = (V_i^j, E_i^j)$ that is induced from the partition V_i^j for $j \in \{1, 2, \ldots, M\}$.

Theorem 8.18. *Given a consistent, acyclic SDF graph $G = (V, E)$, the Ω-Acyclic-Buffering algorithm gives buffer sizes that sustain the maximum achievable throughput in Ω-scheduling over any actor execution time function.*

Proof. Let g_{V_i} denote $\gcd_{v \in V_i} \boldsymbol{q}_G[v]$, and let $v_i^j \in V'_i$ denote the supernode of V_i^j for $j \in \{1, 2, \ldots, M\}$ in G'_i.

Suppose for every $v \in V$, $\boldsymbol{q}_G[v] \times t(v)$ equals a constant value T. First, we assume the buffer sizes of each individual biconnected subgraph $G_i = (V_i, E_i)$ for $i \in \{1, 2, \ldots, N\}$ are set such that each actor $v \in V_i$ can fire continuously at rate $(\boldsymbol{q}_G[v]/g_{V_i})/(T/g_{V_i})$ after a finite transient phase. We refer to this assumption as Assumption (a) . With this Assumption (a) and based on Theorem 8.17, by setting buffer sizes for bridges E_B as in line 2, each actor $v \in V$ can fire continuously at the rate $\boldsymbol{q}_G[v]/T$ after a finite transient phase — this is because by starting with a bridge $[u, v] \in E_B$ or a biconnected component E_i and gradually including adjacent bridges and biconnected components, we can derive by mathematical induction (similar to the proof in Theorem 8.17) that each actor in the graph induced from the current included bridges and biconnected components can fire continuously after a finite transient phase.

Next we show that lines 5-10 make each biconnected subgraph G_i satisfy As-

sumption (a). First, in line 5, a biconnected-free partition $V_i^1, V_i^2, \ldots, V_i^M$ of G_i always exists for $M = 2$ because clustering any *2-way* partition based on a topological sort of an acyclic SDF graph always results in an acyclic two-actor SDF graph, which is free from biconnected components.

We now make another assumption (Assmption b) that by applying Ω- Acyclic-Buffering recursively to each individual subgraph G_i^j for $j \in \{1, 2, \ldots, M\}$, each actor $v \in V_i^j$ can fire continuously at rate $(\boldsymbol{q}_G[v]/g_{V_i^j})/(T/g_{V_i^j})$.

Based on SDF clustering [7], we can derive that for each supernode $v_i^j \in V_i'$, $\boldsymbol{q}_{G_i'}[v_i^j] = g_{V_i^j}/g_{V_i}$, and for each edge $e' \in E'$, e' is transformed from an edge $e \in \{e \in E | src(e) \in V_i^j \text{ and } snk(e) \in V_i^k \text{ and } j \neq k\}$ such that $prd(e') = prd(e) \times \boldsymbol{q}_G[src(e)]/g_{V_i^j}$ and $cns(e') = cns(e) \times \boldsymbol{q}_G[snk(e)]/g_{V_i^k}$. We refer to this property as Property (c) in the remainder of this proof.

Now based on Assumption (b) and Property (c), the execution time for v_i^j can be interpreted as $T/g_{V_i^j}$. Then by setting buffer sizes for E_i' as in line 7 and based on Theorem 8.17, each v_i^j is able to fire continuously after a finite transient phase — we refer to this property as Property (d).

Based on Assumption (b) and Properties (c) and (d), we can derive that Assumption (a) is satisfied for each biconnected subgraph.

Assumptions (a) and (b) are interdependent in the recursive application of the Ω-Acyclic-Buffering algorithm. However, we can always reach a subgraph that contains no biconnected components at the end of the recursion because a biconnected-free partition always exists for an acyclic biconnected graph. As a result, the interdependence of assumptions (a) and (b) can be solved in a recursive way. Therefore, in

the case when $\boldsymbol{q}_G[v] \times t(v)$ equals a constant T for every $v \in V$, Ω-Acyclic-Buffering is able to make each actor $v \in V$ fire continuously at rate $\boldsymbol{q}_G[v]/T = 1/t(v)$ after a finite transient phase, this results in the maximum achievable throughput $1/T$ by Property 8.14 and Theorem 8.8. We refer to this property in the subsequent discussion as Property (e).

As in the proof of Theorem 8.17, we denote $\max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$ by T, and $V_{max} = \{v \in V | \boldsymbol{q}_G[v] \times t(v) = T\}$. Comparing (e) to the case where $\boldsymbol{q}_G[v] \times t(v)$ is not constant across $v \in V$, for an actor $v \in \{V - V_{max}\}$, the smaller $\boldsymbol{q}_G[v] \times t(v)$ does not affect the cycle means of the self loops of V_{max}. Therefore, the Ω-Acyclic-Buffering algorithm gives the maximum achievable throughput $1/\max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$ in Ω-scheduling over any actor execution time function. □

In our implementation of Ω-Acyclic-Buffering, we apply 2-way partitioning for each biconnected subgraph. For efficiency, our approach simply computes a topological sort of a biconnected subgraph and chooses the best 2-way cut that results in least buffer requirements for cross edges (E_i'). With efficient data structures, the operations in Ω-Acyclic-Buffering can be implemented in linear time (i.e., in time that is linear in the number of nodes and edges in the graph) — computing biconnected components and bridges can be done in linear time [16]; given a topological sort, finding the best 2-way cut can be implemented in linear time; and topological sorting has linear time complexity as well [16].

8.4 Compile-Time Scheduling Framework

In the previous section, we introduced Ω-scheduling and associated strategies for throughput analysis and buffering. In this section, we develop compile-time scheduling techniques (including techniques for clustering, ordering, and buffering) based on the Ω-scheduling concept to construct inter-thread communication (ITC) SDF graphs for multithreaded execution.

8.4.1 Clustering and Actor Vectorization

The simplest way to imitate Ω-scheduling in multithreaded environments is to execute each actor by a separate thread and block actor execution until it is bounded-buffer fireable. However, threads share processing resources, and the available resources on a multi-core processor is limited — usually 2 or 4 processing units. As a result, threads are competing for processing units for both execution and synchronization, i.e., checking bounded-buffer fireability. Since the ideal situation is to spend all processing time in actor execution, minimizing synchronization overhead becomes a key factor. In Ω-scheduling, synchronization overhead increases with the repetitions vector of the SDF graph because bounded-buffer fireability must be maintained for every actor firing. Here, we use $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$ to represent the synchronization overhead associated with a consistent SDF graph $G = (V, E)$ in Ω-scheduling.

Clustering combined with static intra-cluster scheduling is one of our strategies to reduce synchronization overhead. After clustering partitions of nodes based on

SDF clustering [7], each cluster is subject to single-thread execution, and the schedule of each cluster (subgraph) is statically computed. We formalize this scheduling strategy in Definition 8.19 and Definition 8.20. The strategy is defined in a general way such that each cluster is assigned to a processing unit (instead of to a thread specifically) and is applicable to scheduling SDF graphs for resource-constrained multiprocessors by controlling the number of partitions.

Definition 8.19. Given a consistent SDF graph $G = (V, E)$, a *consistent partition* P of G is a partition $Z_1, Z_2, \ldots, Z_{|P|}$ of V such that the SDF graph G_P resulting from clustering $Z_1, Z_2, \ldots, Z_{|P|}$ in G is consistent.

Note that actors in a subset Z_i are not necessarily connected. For this reason, we extend the definition of SDF clustering [7] to allow clustering a disconnected subset $Z_i \subset V$ by adding the following provision: if $G_i = (Z_i, E_i = \{e \in E | src(e) \in Z_i \text{ and } snk(e) \in Z_i\})$ is disconnected, $\boldsymbol{q}_{G_i}[v]$ is defined as $\boldsymbol{q}_G[v]/\text{gcd}_{z \in Z_i} \boldsymbol{q}_G[z]$ for each actor $v \in Z_i$.

Definition 8.20. Given a consistent SDF graph $G = (V, E)$, Π-*scheduling* is defined as the SDF scheduling strategy that 1) transforms G into a smaller consistent SDF graph $G_P = (V_P = \{v_1, v_2, \ldots, v_{|P|}\}, E_P)$ by clustering a consistent partition $P = Z_1, Z_2, \ldots, Z_{|P|}$ of G; 2) statically computes a *minimal periodic schedule* S_i for each subgraph $G_i = (Z_i, E_i = \{e \in E | src(e) \in Z_i \text{ and } snk(e) \in Z_i\})$ such that execution of supernode $v_i \in V_P$ corresponds to executing one iteration of S_i; and 3) applies Ω-scheduling on G_P.

After clustering a subset Z_i into a supernode v_i, the repetition count of v_i in G_P

becomes $\boldsymbol{q}_{G_P}[v_i] = \gcd_{v \in Z_i} \boldsymbol{q}_G[v]$. With well-designed clustering algorithm, clustering can significantly reduce synchronization overhead from the range of $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$ down to the range of $Q_{G_P} = \sum_{v \in V_P} \boldsymbol{q}_{G_P}[v]$, but at the expense of buffer requirements and throughput. Clustering may increase buffer requirements because the interface production and consumption rates of the resulting supernodes are multiplied in order to preserve multirate consistency [7]. Clustering also decreases throughput due to less parallelism. In the following theorem, we analyze the clustering effect on the throughput of Ω-scheduling, assuming negligible runtime overhead in executing static schedules and determining bounded-buffer fireability.

Theorem 8.21. *Suppose that we are given a consistent SDF graph $G = (V, E)$, a buffer size function f_B, and an actor execution time function f_T. Suppose also that $P = Z_1, Z_2, \ldots, Z_{|P|}$ is a consistent partition of G and $G_P = (V_P = \{v_1, v_2, \ldots, v_{|P|}\}, E_P)$ is the SDF graph resulting from clustering P. Then a throughput upper bound for G in Π-scheduling, or equivalently, a throughput upper bound for G_P in Ω-scheduling is*

$$\frac{1}{\max_{Z_i \in P}(\sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v)))}. \tag{8.13}$$

In addition, if G_P is acyclic, Equation (8.13) *gives the maximum achievable throughput.*

Proof. Based on Definition 8.20, because each supernode $v_i \in V_P$ is assigned to a separate processing unit, and because firing v_i corresponds to executing one iteration of a minimal periodic schedule of the corresponding subgraph $G_i = (Z_i, E_i)$, the execution time of v_i is $t(v_i) = \sum_{v \in Z_i}(\boldsymbol{q}_{G_i}[v] \times t(v))$. By Theorem 8.10, the throughput

upper bound for G_P in Ω-scheduling is

$$
\begin{aligned}
& 1/\max_{v_i \in V_P}(\boldsymbol{q}_{G_P}[v_i] \times t(v_i)) \\
=& 1/\max_{v_i \in V_P}(\boldsymbol{q}_{G_P}[v_i] \times \sum_{v \in Z_i}(\boldsymbol{q}_{G_i}[v] \times t(v))) \\
=& 1/\max_{Z_i \in P}(\gcd_{z \in Z_i}\boldsymbol{q}_G[z] \times \sum_{v \in Z_i}(\boldsymbol{q}_G[v]/\gcd_{z \in Z_i}\boldsymbol{q}_G[z] \times t(v))) \\
=& 1/\max_{Z_i \in P}(\sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v)))
\end{aligned}
$$

By Theorem 8, Equation (8.13) is the maximum achievable throughput if G_P is acyclic. □

Given a consistent SDF graph $G = (V, E)$, Theorem 8.21 tells us that for clustering a set of actors $Z_i \subseteq V$ into a supernode v_i, a metric that significantly affects the overall throughput is the sum of the repetition count (in terms of G) - execution time products among all actors $v \in Z_i$. For convenience, we denote this value by *SRTP* and define $SRTP(v_i) = SRTP(Z_i) = \sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v))$. Based on Theorem 8.21, the cluster with the largest SRTP value dominates the overall throughput.

In single-processor environments (single-processing unit), the ideal iteration period for executing a consistent SDF graph $G = (V, E)$ is $SRTP(G) = \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$ — that is, the time to execute one iteration of a minimal periodic schedule of G. Now considering an N-core processor (N-processing units), the ideal speed-up over a single-processor is N. In other words, the ideal iteration period on an N-core processor is $\sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))/N$, and equivalently, the ideal throughput is $N/\sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$. In the clustering process, by im-

posing Equation (8.14) as the constraint for each cluster (partition) Z_i, the ideal N-fold speed-up can be achieved theoretically when the SRTP threshold parameter M in Equation (8.14) is greater than or equal to N.

$$\sum_{v \in Z_i} (\boldsymbol{q}_G[v] \times t(v)) \leq \sum_{v \in V} (\boldsymbol{q}_G[v] \times t(v))/M \tag{8.14}$$

In practice, exact actor execution time information is in general unavailable, and execution time estimates may cause large differences between compile-time and run-time SRTP values. As a result, the SRTP threshold parameter M is usually set larger than N in order to tolerate unbalanced runtime SRTP values — that is, by having more small (in terms of compile-time SRTP value) clusters and using multiple threads to share processing units. Based on our experiments, when $N = 4$, the best M is usually between 16 and 32 depending on the graph size and other factors.

Actor vectorization (*actor looping*) is our second strategy to reduce synchronization overhead. Previous work related to actor vectorization in other contexts is discussed in Chapter 3. The main idea in our approach to actor vectorization is to vectorize (loop together) actor executions by a factor of the repetition count of the associated actor. We define actor vectorization as follows.

Definition 8.22. Given a consistent SDF graph $G = (V, E)$, *vectorizing* (*looping*) an actor $v \in V$ by a factor k of $\boldsymbol{q}_G[v]$ means: 1) replacing v by a vectorized actor v^k such that a firing of v^k corresponds to executing v consecutively k times; and 2) replacing each edge $e \in in(v)$ by an edge $e' \in in(v^k)$ such that $cns(e') = cns(e) \times k$, and replacing each edge $e \in out(v)$ by an edge $e' \in out(v^k)$

such that $prd(e') = prd(e) \times k$. For consistency, the *vectorization factor* must be a factor of the repetition count of v. After vectorization, $t(v^k) = t(v) \times k$ and $\boldsymbol{q}_G[v^k] = \boldsymbol{q}_G[v]/k$.

In practical highly multirate SDF graphs, repetitions vectors usually consist of large, non-prime numbers [37]. As a result, actor vectorization is suitable for synchronization reduction in this context, but at the possible expense of larger latency (due to delaying the availability of output tokens in some cases) and larger buffer requirements (due to the multiplication of production and consumption rates). Because we never vectorize an actor beyond its repetition count, and again, because long term simulations require significant numbers of iterations, latency has relatively low priority in our context. Also note that actor vectorization does not change the SRTP value of an actor.

8.4.2 Overview of Compile-Time Scheduling Framework

In our *multithreaded simulation scheduler* (MSS), we have developed a compile-time scheduling framework that jointly performs the clustering, ordering, and buffering tasks as described in Section 8.1 at compile time. In this framework, we strategically integrate several graph clustering and actor vectorization algorithms in a bottom-up fashion such that each subsequent algorithm works on the clustered/vectorized version of the graph from the preceding algorithm. This architecture is presented in Figure 8.7. We also incorporate into this framework intra-cluster scheduling techniques (which include ordering and buffering) as we developed in the

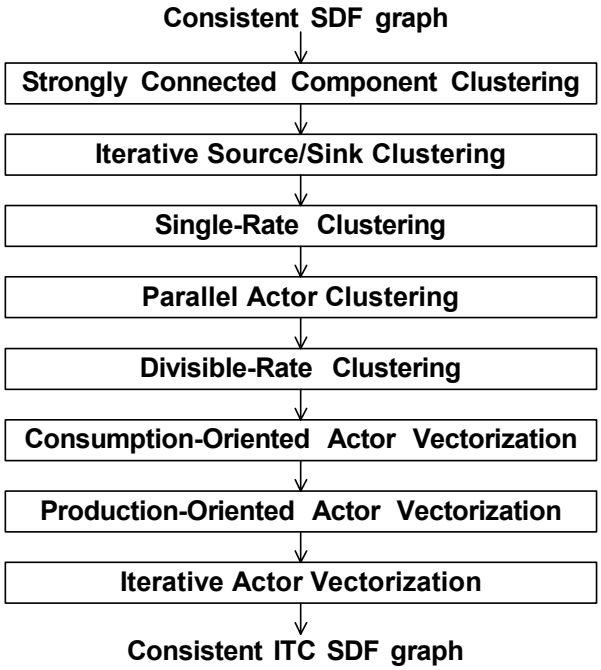


Figure 8.7: Architecture of the clustering and actor vectorization algorithms in MSS.

simulation-oriented scheduler (SOS) (see Chapter 7) such that static intra-cluster schedules (as well as buffer sizes) can be computed along the way in the bottom-up clustering process. Finally, we apply the inter-cluster buffering techniques as presented in Section 8.3.2 to compute buffer sizes for the top-level graph to achieve the maximum achievable throughput theoretically in Ω-scheduling.

Given a consistent SDF graph $v \in V$ as input to the compile-time scheduling framework, the resulting graph $G_{itc} = (V_{itc}, E_{itc})$ is called an *inter-thread communication* (ITC) SDF graph (or simply *ITC graph*) because each node (cluster) in G_{itc} is executed by a thread. The ITC graph is then passed to the runtime scheduling part of MSS for multithreaded execution (see Section 8.5). In MSS, ITC graphs are carefully constructed from input SDF graphs for proper trade-offs among the following three related metrics:

1. Synchronization overhead — Synchronization overhead is reduced by clustering and actor vectorization such that the repetitions vector of the clustered/vectorized SDF graph is much smaller, and hence, the time spent in checking bounded-buffer fireability is minimized.
2. Throughput — Based on Theorem 8.21, clustering decreases throughput. In order to approach the ideal throughput in a multi-core processor, Equation (8.14) is imposed in the clustering process given a well-defined SRTP threshold parameter M. As a result, the SRTP value of each resulting cluster in G_{itc} is kept below the *SRTP threshold* — that is, $\sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))/M$ with respect to the input graph G — if possible.
3. Buffer requirements — Simulation tools usually run on workstations and high-end PCs where memory resources are abundant. However, without careful design, clustering and actor vectorization may still run-out of memory due to large multirate complexity [37]. In our approach, total buffer requirements are managed within the given buffer upper bound. A proper *buffer memory upper bound* can be derived from the available memory resources in the simulation environment and other relevant considerations.

In this framework, all of the integrated algorithms emphasize low complexity for minimizing the time spent in compile-time scheduling. In addition, clustering algorithms are carefully designed to prevent introduction of cycles in the clustered version of the graph. This is because cycles may cause deadlock due to cyclic data dependence. Furthermore, even without deadlock, cycles may cause limitations in the maximum achievable throughput. The following theorem provides a precise

condition for the introduction of a cycle by a clustering operation. The proof can be found in Section 7.3.5.

Theorem 8.23. *Given a connected, acyclic SDF graph $G = (V, E)$, clustering a subset $R \subseteq V$ introduces a cycle in the clustered version of G if and only if there is a path $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ ($n \geq 3$) in G such that $v_1 \in R$, $v_n \in R$, and $v_2, \ldots, v_{n-1} \in \{V - R\}$. Clustering R is cycle-free if and only if no such a path exists.*

In the following subsections, we introduce our algorithms for clustering and actor vectorization. The complexity of an algorithm is represented in terms of the number of vertices $|V|$ and edges $|E|$ in the input graph $G = (V, E)$ for each individual algorithm (*not* the overall graph to the scheduling framework). Based on this, $|V|$ and $|E|$ (the input sizes for the various algorithms) get progressively smaller through the bottom-up clustering process. For complexity analysis, we make the assumption that every actor has a constant (limited) number of input and output edges, i.e., $|V|$ and $|E|$ are within a similar range. This is a reasonable assumption because actors in simulation tools are usually pre-defined, and practical SDF graphs in communications and signal processing domains are sparse in their topology [7].

Our clustering and actor vectorization algorithms extensively use the iterative approach, i.e., applying the same operations iteratively to the clustered or vectorized version of the graph from the previous iteration. The following operations and complexity analysis concepts are common to our algorithms. First, the repetitions vector of the overall graph can be computed in *linear time* (i.e., in time that is linear in the number of nodes and edges in the graph) [7], and once the repetitions vector

has been computed, the SRTP value of each individual actor can be obtained in constant time. Second, suppose $G = (V, E)$ is an input SDF graph to an individual algorithm. Based on SDF clustering [7], clustering a subset of actors $Z \subseteq V$ into a supernode α takes $O(|Z|)$ running time. Suppose now that G' is the graph that results from the aforementioned clustering operation. Then the repetition count and SRTP value of supernode α in G' can be computed by $\boldsymbol{q}_{G'}[\alpha] = \gcd_{v \in Z} \boldsymbol{q}_G[v]$ and $SRTP(\alpha) = \sum_{v \in Z} SRTP(v)$. As a result, repetition counts and SRTP values of supernodes are available after clustering operations.

8.4.3 Strongly Connected Component Clustering

According to Theorem 8.10, the existence of cycles in an ITC SDF graph may decrease the maximum achievable throughput depending on the locations and magnitudes of edge delays in those cycles. Moreover, the presence of cycles restricts application of many useful scheduling techniques in our framework. Clustering strongly connected components [1] (SCCs) is a well-known technique to generate acyclic graphs [16]. Based on our analysis, if the SRTP value of each SCC satisfies Equation (8.14), clustering SCCs does not cause limitations in the achievable throughput.

Given a consistent SDF graph, we first apply *strongly connected component clustering* to cluster all SCCs, and this results in an acyclic SDF graph to be further

[1] A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $Z \subseteq V$ such that for every pair of vertices u and v in Z, there is a path from u to v and a path from v to u.

processed by the subsequent algorithms. Each SCC subgraph is then scheduled efficiently by the *simulation-oriented scheduler* (see Chapter 7). SCC clustering has linear-time complexity — computing SCCs of a directed graph can be implemented in linear time [16], and the complexity of clustering all SCCs is bounded by the number of nodes and edges in the graph.

8.4.4 Iterative Source/Sink Clustering

In practical communication and signal processing systems, subsystems having the form of chain- or tree-structures arise frequently. Based on Theorem 8.23, clustering such subsystems at the source-end or sink-end does not introduce cycles because there is only one connection between the subsystems and the rest of the graph. In addition, if the SDF production and consumption rate (data rate) behavior involved in such a subsystem is successively divisible in certain ways, then clustering such a subsystem does not increase the production or consumption rates of the resulting supernode. Figure 8.8 illustrates our targeted subsystems. The idea of *iterative source/sink clustering* (ISSC) is to jointly explore the chain- or tree-structures and the successively divisible rate behavior in a low-complexity manner such that clustering is always cycle-free and does not increase the buffer requirements.

Here, we first define some notation that is useful to our development. Given a directed graph $G = (V, E)$, we say that a vertex $v \in V$ is a *source* if v does not have any input edges ($in(v) = \varnothing$), and is a *sink* if v does not have any output edges

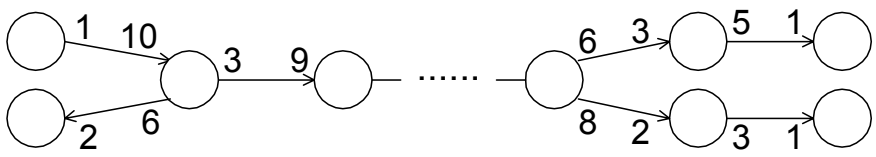


Figure 8.8: Examples of targeted subsystems in ISSC.

($out(v) = \varnothing$). For an edge $e \in E$, we say that $src(e)$ is a *predecessor* of $snk(e)$, and $snk(e)$ is a *successor* of $src(e)$. For a vertex v, we denote all of v's predecessors by $pre(v)$, denote all of v's successors by $suc(v)$, and denote all of v's adjacent actors by $adj(v) = \{pre(v) + suc(v)\}$. We then define the ISSC technique as follows.

Definition 8.24. Given a consistent, acyclic SDF graph $G = (V, E)$, the *iterative source/sink clustering* (ISSC) technique *iteratively* clusters:

1) a source actor u with its successor if 1-a) u has one and only one successor v, 1-b) $\boldsymbol{q}_G[u]$ is divisible by $\boldsymbol{q}_G[v]$, and 1-c) $SRTP(u) + SRTP(v)$ is less than or equal to the SRTP threshold; or
2) a sink actor v with its predecessor if 2-a) v has one and only one predecessor u, 2-b) $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[u]$, and 2-c) $SRTP(u) + SRTP(v)$ is less than or equal to the SRTP threshold.

Clustering based on these conditions continues until there is no further clustering can be performed. After each iteration, G represents the clustered version of the graph that is subject to the next iteration.

Note that condition 1-b in Definition 8.24 can also be interpreted as: $\forall e \in [u, v]$, $cns(e)$ is divisible by $prd(e)$; and similarly, condition 2-b can be interpreted as: $\forall e \in [u, v]$, $prd(e)$ is divisible by $cns(e)$. Because of divisible data rates, each two-

```
ISSC(G)
input: a consistent acyclic SDF graph G = (V, E)
1   for the next actor v in G
    /* the next actor refers to the supernode from the previous iteration or
    the next actor that has not yet been visited in the current G */
2       if v satisfies condition 1 or condition 2 in Definition 8.24
3           cluster v and its adjacent actor
4       end
5   end
```

Figure 8.9: Iterative source/sink clustering algorithm.

actor cluster constructed in ISSC iterations can be scheduled efficiently by *flat scheduling* (see Section 7.3.6).

Figure 8.9 presents the *ISSC* algorithm. In the *for* loop in line 1, "the next actor" represents the new supernode from the previous iteration if the previous iteration has performed clustering, or the next actor that has not yet been visited in the current version of G. By this definition, the number of actors processed by the *for* loop is bounded by $O(|V|)$. With efficient data structures, obtaining "the next actor" can be performed in constant time, and verifying qualification of an actor for Definition 8.24 also takes constant time. As a result, the ISSC algorithm has linear-time complexity.

8.4.5 Single-Rate Clustering

Intuitively, a single-rate subsystem in an SDF graph is a subsystem in which all actors execute at the same average rate. In precise terms, an SDF graph is a *single-rate* graph if for every edge e, we have $prd(e) = cns(e)$. In practical communication and signal processing systems, single-rate subsystems arise commonly, even within designs that are heavily multirate at a global level. Since clustering single-rate

subsystems does not increase production and consumption rates at the interface of the resulting supernodes, we integrate the *single-rate clustering* (SRC) technique in our framework. SRC has been developed in the simulation-oriented scheduler (see Chapter 7), and Section 7.3.5 presents the SRC algorithm, which takes $O(|E|^2)$ time. Here, we define SRC again as follows. For the associated algorithm and theorems, we refer the reader to Section 7.3.5.

Definition 8.25. Given a connected, consistent, acyclic SDF graph $G = (V, E)$, the *single-rate clustering* (SRC) technique clusters disjoint subsets $R_1, R_2, \ldots, R_N \subseteq V$ such that: 1) in the subgraph $G_i = (R_i, E_i)$, we have that $\forall e_i \in E_i = \{e \in E \mid src(e) \in R_i \text{ and } snk(e) \in R_i\}$, $prd(e_i) = cns(e_i)$; 2) the clustering of R_i does not introduce any cycles into the clustered version of G; 3) R_i satisfies $|R_i| > 1$ (i.e., R_i contains at least two actors); and 4) each R_i contains a maximal set of actors that satisfy all of the three conditions above. Such R_is are defined as *single-rate subsets*; and such G_is are defined as *single-rate subgraphs*.

In addition to Definition 8.25, in order to maintain the achievable throughput in multithreaded execution, a single-rate subset R_i whose SRTP value is larger than the SRTP threshold is further partitioned into multiple *single-rate clusters* such that each cluster satisfies Equation (8.14) whenever possible. By partitioning R_i based on a topological sort[2], clustering single-rate clusters is cycle-free. This additional process requires only linear time because topological sorts can be computed in linear time [16]. Due to their simple structure, single-rate subgraphs (clusters) can be

[2]A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of V such that for every edge (u, v) in G, u appears before v in the ordering.

statically scheduled and optimized effectively by flat scheduling [7] in linear time, which simply computes a topological sort and iterates each actor by its repetition count. For more details, we refer the reader to Section 7.3.5.

8.4.6 Parallel Actor Clustering

In practical communication and signal processing systems, subsets of *parallel actors* often exist. Here, we say actors u and v are *parallel* if they are *mutually unreachable* [3] — that is, there is no path from u to v nor from v to u. According to Theorem 8.23, clustering parallel actors is cycle-free because there is no path between any pair of parallel actors. In addition, based on our extended definition of SDF clustering, clustering parallel actors with the same repetition count does not increase the production and consumption rates of the resulting supernode.

In general, computing reachability information of a directed graph requires $\Theta(|V|^3)$ time by the Floyd-Warshall algorithm [16], and updating reachability information after a clustering operation requires at least $O(|V|^2)$ time. However, the complexity of this process is too high to satisfy our goal in minimizing scheduling runtime. Moreover, arbitrarily clustering parallel actors often introduces new biconnected components or expands existing biconnected components. Figure 8.10 illustrates such an example: actors d and e are parallel and have the same repetition count, but clustering $\{d, e\}$ introduces a biconnected component. Based on Ω-Acyclic-Buffering, introduction and expansion of biconnected components generally complicates the buffer computation problem and may increase the overall buffer

[3] Given a directed graph, a vertex v is reachable from u if there is a path from u to v.

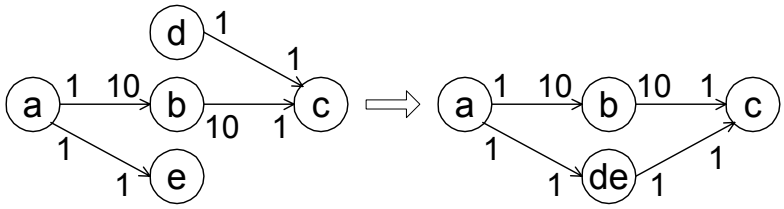


Figure 8.10: Introduction of a biconnected component due to clustering.

requirements.

The idea of *parallel actor clustering* is to jointly explore parallel structures and associated repetition count properties such that clustering does not increase the production and consumption rates of the resulting supernodes and does not introduce cycles nor biconnected components.

Here, we first present a *topological ranking* technique as shown in Figure 8.11 that helps us to explore certain parallel structures in linear running time. This technique is motivated by graph drawing technique employed in DOT [23]. Given a directed acyclic graph $G = (V, E)$, the idea of topological ranking is to assign an integer value (*rank*) $r(v)$ to each vertex $v \in V$ such that for each edge $e \in E$, we have $r(snk(e)) > r(src(e))$, and vertices with the same rank are parallel.

Property 8.26. *Suppose the topological ranking algorithm (as presented in Figure 8.11) is applied to a directed acyclic graph, then vertices with the same rank are parallel.*

Note that parallel actors may not have the same rank. Topological ranking is primarily developed as a low-complexity approach to explore certain parallel structures in directed acyclic graphs. With the above technique, we present the parallel

```
TOPOLOGICAL-RANKING(G)
input: a directed acyclic graph G = (V, E)
 1  for v ∈ V   r(v) = −∞   end
 2  list L = TOPOLOGICAL-SORT(G)
 3  pop the first vectex v from L, set r(v) = 0
 4  while L is not empty
 5     for v ∈ L in the forward direction
 6        P = {α ∈ pre(v)|r(α) ≠ −∞}
 7        if P ≠ ∅   r(v) = max_{α∈P} r(α) + 1, remove v from L   end
 8     end
 9     for v ∈ L in the reverse direction
10        S = {α ∈ suc(v)|r(α) ≠ −∞}
11        if S ≠ ∅   r(v) = min_{α∈S} r(α) − 1, remove v from L   end
12     end
13  end
```

Figure 8.11: Topological ranking algorithm.

actor clustering technique as follows.

Definition 8.27. Given a consistent, acyclic SDF graph $G = (V, E)$ and a topological rank of G, the *parallel actor clustering* (PAC) technique *iteratively* clusters a set of actors R ($|R| > 1$) that satisfy the following conditions until no further clustering can be made:

1) all actors in R have the same rank;

2) all actors in R have the same repetition count;

3) {(all actors in R have the same predecessor v) and (the edges between v and R, $E_{v,R} = \{e|src(e) = v \text{ and } snk(e) \in R\}$, belong to the same biconnected component or $E_{v,R}$ are bridges)} or

   {(all actors in R have the same successor u) and (the edges between R and u, $E_{R,u} = \{e|src(e) \in R \text{ and } snk(e) = u\}$, belong to the same biconnected component or $E_{R,u}$ are bridges)}; and

4) $SRTP(R)$ is less than or equal to the SRTP threshold.

Such R is defined as a *parallel actor subset*. After each iteration, the resulting supernode inherits the same rank, and G represents the clustered version of the graph that is subject to the next iteration.

In Definition 8.27, condition 3 prevents introduction of biconnected components. Scheduling for a parallel actor subset R is trivial. Because precedence constraints do not exist in R, and since actors in R have the same repetition count, a static schedule can be easily constructed by firing each actor once in any order. Buffering is not required because parallel actor subgraphs do not contain edges.

Figure 8.12 presents the *PAC* algorithm. In Figure 8.12, we first compute a topological ranking for G in linear time. For fast implementation, we also compute biconnected components $E_1, E_2, \ldots, E_N$ of G in advance. Then for each edge e in a biconnected component E_i, we assign an identifier $b(e) = i$, and for each bridge edge e, we assign $b(e) = 0$. This process can also be done in linear time. Now given $pre(v)$ in line 4, parallel actor subsets can be computed by 1) sorting and partitioning $pre(v)$ based on the same rank values, the same repetition counts, and the same biconnected component identifiers of the corresponding edges to v, and then 2) partitioning each resulting subset if its SRTP value is larger than the SRTP threshold. With efficient data structures, computing parallel actor subsets from $pre(v)$ can be implemented in $O(|pre(v)| \log(|pre(v)|))$ time. We apply the same approach for $suc(v)$ in line 7. After clustering a parallel actor subset R into a supernode α, α can inherit the same rank and the same repetition count, and the resulting input and output edges of α can inherit the same biconnected component

```
PAC(G)
input: a consistent acyclic SDF graph G = (V, E)
 1  TOPOLOGICAL-RANKING(G)
 2  {E_1, E_2, ..., E_N} = BICONNECTED-COMPONENTS(G)
 3  for the next actor v in G
    /* the next actor refers to the next actor that has not yet been visited in the
    current version of G including the supernodes from the previous iterations */
 4     if pre(v) contains parallel actor subsets R_1, R_2, ..., R_m
 5        cluster R_1, R_2, ..., R_m
 6     end
 7     if suc(v) contains parallel actor subsets Z_1, Z_2, ..., Z_n
 8        cluster Z_1, Z_2, ..., Z_n
 9     end
10  end
```

Figure 8.12: Parallel actor clustering algorithm.

identifiers without further computation.

In the for loop in line 3, "the next actor" refers to the next actor that has not yet been visited in the current version of G, which includes the newly constructed supernodes from previous iterations. Again, with efficient data structures, obtaining "the next actor" can be done in constant time, and the number of actors processed by the *for* loop is bounded by $O(|V|)$. As discussed in Section 8.4.2, we assume every actor has limited number of input and output edges. Therefore, the PAC algorithm has linear-time complexity $O(|E|)$.

Based on our experiments, parallel actor clustering is very effective at clustering parallel structures such as that shown in Figure 8.13. In some well-structured cases, parallel actor clustering can even eliminate biconnected components.

8.4.7 Divisible-Rate Clustering

In practical communication and signal processing systems, data rate behavior associated with an actor and its surrounding actors possesses certain valuable prop-

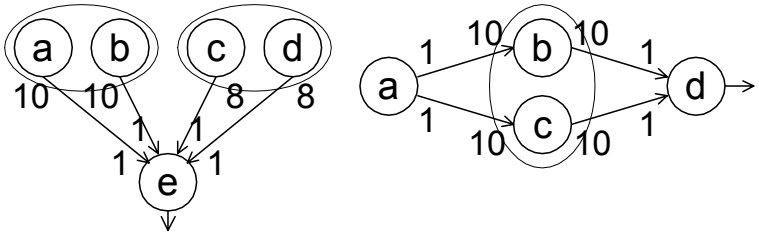


Figure 8.13: Parallel actor clustering examples.

erties that can be explored in our clustering framework. For example, single-rate clustering exploits single-rate subsystems. In this section, we present the *divisible-rate clustering* technique to explore both single-rate and multirate behavior of an actor in relation to its adjacent actors such that buffer requirements can be maintained after clustering.

Definition 8.28. Given a consistent, acyclic SDF graph $G = (V, E)$, the *divisible-rate clustering* (DRC) technique *iteratively* clusters an actor v with one of its candidate adjacent actors $u \in adj(v)$ such that:

1) either 1-a) $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$ or 1-b) for every $x \in adj(v)$, $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[x]$, and $\boldsymbol{q}_G[u]$ is divisible by $\boldsymbol{q}_G[x]$;
2) $SRTP(v) + SRTP(u)$ is less than or equal to the SRTP threshold; and
3) clustering $\{v, u\}$ is cycle-free.

This clustering process continues until no further clustering can be performed. After each iteration, G represents the clustered version of the graph that is subject to the next iteration.

In the divisible-rate clustering process, a candidate pair of adjacent actors $\{v, u\}$ is carefully chosen in each iteration. Suppose $\{v, u\}$ satisfies condition 1-a

in Definition 8.28, then clustering $\{v, u\}$ does not increase the interface data rates of the resulting supernode. Suppose $\{v, u\}$ satisfies condition 1-b and suppose α represents the resulting supernode, then based on SDF clustering [7], for an edge $e \in in(\alpha)$, if $cns(e)$ has been increased due to clustering, then $prd(e)$ is divisible by $cns(e)$, and for an edge $e \in out(\alpha)$, if $prd(e)$ has been increased, then $cns(e)$ is divisible by $prd(e)$. In many intra-cluster scheduling and inter-cluster buffering scenarios, this divisible-rate property maintains the buffer requirements of input and output edges for the resulting supernodes. In addition, the two-actor cluster $\{v, u\}$ can be scheduled efficiently by the flat scheduling [7] because $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[u]$.

Figure 8.14 presents the *DRC* algorithm. Verifying a target actor v and finding a candidate adjacent actor u of v for both condition 1 and condition 2 in Definition 8.28 take $O(|adj(v)|)$ time. For condition 3, we have developed an algorithm in Section 7.3.5 to determine whether clustering a pair of adjacent actors is cycle-free based on Theorem 8.23, and this algorithm takes $O(|V| + |E|)$ time. Again, as with Figure 8.9, obtaining "the next actor" in the *for* loop in line 1 can be performed in constant time, and the number of actors processed by the *for* loop is bounded by $O(|V|)$. As a result, the complexity of the DRC algorithm is $O(|E|^2)$.

8.4.8 Consumption-/Production-Oriented Actor Vectorization

Modeling real-world communication and signal processing systems as SDF graphs usually involves large numbers of components that are interconnected in

```
DRC(G)
input: a consistent acyclic SDF graph G = (V, E)
1   for the next actor v in G
    /* the next actor refers to the supernode from the previous iteration or
    the next actor that has not yet been visited in the current G */
2       if there exists an adjacent actor u of v that satisfy Definition 8.28
3           cluster {v, u}
4       end
5   end
```

Figure 8.14: Divisible-rate clustering algorithm.

complex topologies, and have heavily multirate behavior. The techniques introduced from Section 8.4.4 to Section 8.4.7 are effective in clustering such SDF graphs (for synchronization reduction) while maintaining the buffer requirements. The resulting SDF graphs in general have significantly smaller sizes and contain mixes of single-rate and multirate edges. The remaining single-rate edges are largely due to the SRTP threshold that prevents us from completely clustering single-rate subsets.

Here, we define an SDF edge e to be *rate-indivisible* if $prd(e)$ and $cns(e)$ are mutually indivisible; to be *production-rate-divisible* if $prd(e)$ is divisible by $cns(e)$; and to be *consumption-rate-divisible* if $cns(e)$ is divisible by $prd(e)$. By further investigation of multirate behavior in the practical SDF graphs, we have observed that subsystems containing neighboring, production-rate-divisible or consumption-rate-divisible interconnections often exist in highly multirate systems, and these subsystems may also interact with rate-indivisible interconnections.

In the remainder of this section, we focus on actor vectorization techniques. Given a consistent, acyclic SDF graph $G = (V, E)$, the ideal situation for actor vectorization is to vectorize each actor $v \in V$ by its repetition count $\boldsymbol{q}_G[v]$, and have the total buffer requirement of the vectorized version of G remain within the given

upper bound. Again, in our context, a proper buffer memory upper bound can be derived from the available memory resources and other relevant considerations. In this case, the resulting ITC graph is just a single-rate SDF graph, and the synchronization overhead is reduced to the range of $|V|$. However, due to large-scale and heavily multirate behavior involved in modern communication and signal processing systems, the ideal situation may not happen in general.

For this purpose, we develop the *consumption-oriented actor vectorization* (CAV) technique and the *production-oriented actor vectorization* (PAV) technique in this subsection and the *iterative actor vectorization* (IAV) technique in the following subsection to strategically trade off buffer cost for synchronization reductions. In the vectorization process, the total buffer requirement is carefully kept under control within the given upper bound to prevent out-of-memory problems. Given a buffer computation function f_B for G, we use $f_I(G, f_B, v \rightarrow v^k)$ to denote the increase in buffer requirements when vectorizing an actor v by a factor k. This notation is important to our developments of actor vectorization techniques.

By Definition 8.22, a vectorization factor must be a factor of the actor's repetition count in order to maintain graph consistency. However, heavily multirate systems often result in extremely high repetition counts, e.g., even up to the range of millions, as we show in Section 7.5. As a result, the complexity to determine optimal vectorization factors is in general unmanageable in highly multirate systems. In this subsection, we use divisible multirate properties associated with an actor v and its adjacent actors to determine possible vectorization factors — that is, for an adjacent actor u of v, if $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[u]$, then $\boldsymbol{q}_G[v]/\boldsymbol{q}_G[u]$ is considered

as a vectorization factor for v.

The idea of our CAV technique is to take advantage of consumption-rate-divisible edges for actor vectorization and to explore single-rate clustering opportunities exposed by such actor vectorizations. CAV favors latency because the source actor of a consumption-rate-divisible edge is vectorized to match the rate of the sink actor. The design of CAV also prevents propagation of indivisible rates — without careful design, such propagation may cause larger buffer requirements and reduce opportunities for proper clustering. Here, we present the CAV technique as follows.

Definition 8.29. Suppose that we are given a consistent, acyclic SDF graph $G = (V, E)$, a buffer computation function f_B, and a buffer memory upper bound U. The *consumption-oriented actor vectorization* (CAV) technique *iteratively* selects an actor v for vectorization and clustering until there no further vectorization can be performed or the total buffer requirement approaches the upper bound U. An actor $v \in V$ is considered as a *candidate* for vectorization if:

1) for every $x \in suc(v)$, $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[x]$;
2) there exists an actor $u \in suc(v)$ such that for every $x \in suc(v)$, $\boldsymbol{q}_G[u]$ is divisible by $\boldsymbol{q}_G[x]$; and
3) the buffer cost increase $f_I(G, f_B, v \rightarrow v^k)$ resulting from vectorizing v by a factor $k = \boldsymbol{q}_G[v]/\boldsymbol{q}_G[u]$ does not overflow the upper bound U.

In each iteration, CAV selects a candidate actor v whose repetition count $\boldsymbol{q}_G[v]$ is maximal over all candidates and then vectorizes v by the factor k. After vectorization, CAV *iteratively* clusters v with its adjacent actor $u \in adj(v)$ if:

a) $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$ (single-rate);

b) $SRTP(v) + SRTP(u)$ is less than or equal to the SRTP threshold; and

c) clustering $\{v, u\}$ is cycle-free.

After each actor vectorization and clustering iteration, G represents the vectorized or clustered version of the graph that is subject to the next iteration.

In Definition 8.29, conditions 1 and 2 prevent propagation of indivisible rates, and condition 3 prevents buffer overflow. After each clustering operation, the two-actor cluster $\{v, u\}$ can be scheduled efficiently by flat scheduling [7] because $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$.

Figure 8.15 presents the *CAV* algorithm. In our implementation within MSS, we first apply the Ω-Acyclic-Buffering algorithm as f_B to compute buffer requirements, and we also keep the hierarchical structures constructed from the decompositions of biconnected components in Ω-Acyclic-Buffering. By maintaining the hierarchical structures, the buffer requirements for bridge edges (E_B in Figure 8.6) can be computed in constant time based on the updated production and consumption rates, and the buffer requirements for edges across clusters (E_i' in Figure 8.6) can also be computed in constant time based on the updated production and consumption rates and the updated repetition counts of the clusters.

Given a consistent, acyclic SDF graph $G = (V, E)$, based on Definition 8.29, the complexity to verify an actor v as a candidate and to determine the vectorization factor k is $O(|suc(v)|)$. In each iteration, the running time to select a candidate actor v whose repetition count is maximal over all candidates is bounded

```
CAV(G,f_B,U)
input: a consistent acyclic SDF graph G = (V, E),
    a buffer computation function f_B, and a buffer memory upper bound U
1   compute buffer sizes by f_B
2   while there exists a repetition-count-maximal, candidate actor v based on Definition 8.29
3       vectorize v by the vectorization factor k determined by Definition 8.29
4       iteratively cluster v with its adjacent actors adj(v) based on Definition 8.29
5       update buffer sizes
6   end
```

Figure 8.15: Consumption-oriented actor vectorization algorithm.

by $O(\sum_{v \in V} |suc(v)|) = O(|E|)$. In addition, the complexity involved in lines 3-5 is bounded by $O(|E|)$. As a result, the complexity of an actor vectorization iteration is $O(|E|)$. Finally, consider the worst case situation where the repetition counts for every pair of actors in V are divisible, and each actor repetition count in $\boldsymbol{q}_G$ is a unique value. Then an actor can be selected for vectorization in a maximum of $|V|$ different iterations. Therefore, the complexity of the joint actor vectorization and clustering technique is bounded by $O(|V|^2|E|)$.

At first, this complexity appears relatively high in relation to the objective of low complexity. However, due to the design of our compile-time scheduling framework, which applies a series of clustering techniques (as described in Section 8.4.3 to Section 8.4.7), $|V|$ and $|E|$ are typically much smaller compared to the numbers of actors and edges in the overall SDF graph.

The PAV technique is similar to CAV, but PAV focuses on production-rate-divisible edges. In MSS, CAV is applied before PAV because buffer cost can be traded off for both synchronization overhead and latency by CAV.

8.4.9 Iterative Actor Vectorization

The actor vectorization techniques discussed in the previous subsection are able to explore both actor vectorization and clustering opportunities in subsystems that contain consecutive, consumption-rate-divisible or production-rate-divisible interconnections. In this subsection, we present a general actor vectorization approach, called *iterative actor vectorization* (IAV). This approach trades off buffer cost for synchronization cost, and also handles indivisible multirate interconnections.

As discussed in Section 8.4.1, we use $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$ to represent synchronization overhead associated with a consistent SDF graph $G = (V, E)$ in Ω-scheduling. Based on this representation, after vectorizing an actor $v \in V$ by a factor k of $\boldsymbol{q}_G[v]$, the amount of synchronization reduction can be represented by $\boldsymbol{q}_G[v](1 - 1/k)$. Then, a general strategy is to vectorize a properly-chosen actor by a well-determined factor such that the synchronization reduction can be maximized while the penalty in buffer cost is minimal. Based on this observation, we define the *synchronization reduction to buffer increase ratio* (or simply *S/B ratio*) —

$$R_{S/B} : \frac{\boldsymbol{q}_G[v](1 - 1/k)}{f_I(G, f_B, v \rightarrow v^k)} \tag{8.15}$$

as the cost function for actor vectorization.

As discussed in the previous subsection, the complexity of finding optimal vectorization factors is in general unmanageable in highly multirate systems. In this approach, we use multirate behavior associated with an actor v and its adjacent actors to determine candidate vectorization factors — that is, for an adjacent actor u of v, if $\boldsymbol{q}_G[v] > \boldsymbol{q}_G[u]$, then $\boldsymbol{q}_G[v]/\text{gcd}(\boldsymbol{q}_G[v], \boldsymbol{q}_G[u])$ is considered as a vectorization

factor for v.

Based on the above derivations, we develop the iterative actor vectorization technique as follows.

Definition 8.30. Suppose that we are given a consistent, acyclic SDF graph $G = (V, E)$, a buffer computation function f_B, and a buffer memory upper bound U. The *iterative actor vectorization* (IAV) technique *iteratively* vectorizes an actor v by a factor k of $\boldsymbol{q}_G[v]$ until there no further vectorization can be performed or the total buffer requirement approaches the upper bound U. An actor $v \in V$ is considered for vectorization if v is a *local maximum* — that is,

1) for every adjacent actor $u \in adj(v)$, $\boldsymbol{q}_G[v] \geq \boldsymbol{q}_G[u]$, and

2) there exists at least one adjacent actor $u \in adj(v)$ such that $\boldsymbol{q}_G[v] > \boldsymbol{q}_G[u]$.

For such a local maximum actor v, the vectorization factor k is determined from the factors

$$\left\{ \frac{\boldsymbol{q}_G[v]}{\gcd(\boldsymbol{q}_G[v], \boldsymbol{q}_G[u])}, \forall u \in \{u \in adj(v) | \boldsymbol{q}_G[u] < \boldsymbol{q}_G[v]\} \right\} \tag{8.16}$$

such that the S/B ratio is maximized — that is, we maximize

$$R_{S/B} : \frac{\boldsymbol{q}_G[v](1 - 1/k)}{f_I(G, f_B, v \rightarrow v^k)}$$

subject to the constraint that the buffer cost increase $f_I(G, f_B, v \rightarrow v^k)$ does not overflow the upper bound U. In each iteration, a local maximum actor v is chosen such that the S/B ratio is maximized for v over all local maximum actors. After vectorization, IAV *iteratively* clusters v with its adjacent actor $u \in adj(v)$ if:

a) $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$ (single-rate);

b) $SRTP(v) + SRTP(u)$ is less than or equal to the SRTP threshold; and

IAV(G,f_B,U)
input: a consistent acyclic SDF graph $G = (V, E)$,
a buffer computation function f_B, and a buffer memory upper bound U
objective: actor vectorization on G based on Definition 8.30
1 compute buffer sizes by f_B
2 **while** there exists an actor v and a vectorization factor k
based on Definition 8.30 to maximize the S/B ratio for G
3 vectorize v by k
4 iteratively cluster v with its adjacent actors $adj(v)$ based on Definition 8.30
5 update buffer sizes
6 **end**

Figure 8.16: Iterative actor vectorization algorithm.

c) clustering $\{v, u\}$ is cycle-free.

After each iteration, G represents the vectorized version of the graph that is subject to the next iteration.

Figure 8.16 presents the IAV algorithm as defined in Definition 8.30. We apply the same buffering approach f_B as described in Section 8.4.8. Given a consistent, acyclic SDF graph $G = (V, E)$, based on Definition 8.30, the number of possible vectorization factors of an actor $v \in V$ is bounded by O($|adj(v)|$). For a vectorization factor, the running time to compute the corresponding S/B ratio is O($|adj(v)|$) — this is because vectorizing v may increase the buffer sizes of its input and output edges, and these increases should all be taken into account in determining the buffer cost increase f_I. As a result, the complexity to compute a vectorization factor that maximizes the S/B ratio for v is O($|adj(v)|^2$).

In each iteration, considering the worst case situation where all actors are local maxima, the complexity to jointly determine an actor and a vectorization factor that maximizes the S/B ratio for G is O($\sum_{v \in V} |adj(v)|^2$). Let A denote the maximum number of adjacent actors that an actor can have in G and the clustered versions

of G. Because $O(\sum_{v \in V} |adj(v)|) = O(|E|)$, we can represent $O(\sum_{v \in V} |adj(v)|^2)$ by $O(|E|A)$. Again, as discussed in Section 8.4.2, we assume every actor has limited number of input and output edges — that is, for large G, A can be considered as a constant. In addition, the complexity involved in lines 3-5 is bounded by $O(|E|)$. As a result, the complexity of an actor vectorization iteration is $O(|E|)$. Finally, considering the worst case situation that each actor repetition count in $\boldsymbol{q}_G$ is a unique value, each actor can be selected for vectorization in a maximum of $|V|$ different iterations. Therefore, the complexity of iterative actor vectorization is $O(|V|^2|E|)$.

8.5 Runtime Scheduling

In the previous section, we introduced the compile-time scheduling framework in MSS. This framework integrates graph clustering, actor vectorization, intra-cluster scheduling, and inter-cluster buffering techniques to construct inter-thread communication (ITC) SDF graphs. In this section, we develop runtime scheduling techniques for the assignment and synchronization tasks in scheduling ITC graphs for multithreaded execution.

As discussed in Section 8.4, given a consistent SDF graph $G = (V, E)$, the compile-time scheduling framework constructs an ITC graph $G_{itc} = (V_{itc}, E_{itc})$ for multithreaded execution. Here, we refer to vertices and edges in G_{itc} as *ITC nodes* and *ITC edges*. An ITC node $v \in V_{itc}$ represents either 1) an actor in G or 2) a cluster of actors in G that is constructed during the clustering process. Firing v *once* means executing either 1) the actor or 2) the static schedule (by intra-cluster

scheduling) of the cluster for a *specific vectorization factor* that is determined during the actor vectorization process. For each ITC edge $e \in E_{itc}$, its buffer size $buf(e)$ must also be set during the inter-cluster buffering process (Ω-Acyclic-Buffering) for G_{itc}.

8.5.1 Self-Timed Multithreaded Execution Model

In the following definition, we develop the *self-timed multithreaded execution model* to imitate Ω-scheduling for executing ITC graphs in multithreaded environments.

Definition 8.31. Given a consistent ITC SDF graph $G_{itc} = (V_{itc}, E_{itc})$, the *self-timed multithreaded execution model* allocates a number of threads equal to the number of ITC nodes $|V_{itc}|$ and assigns each ITC node $v \in V_{itc}$ to a separate thread. Each thread executes the associated ITC node v as soon as v is bounded-buffer fireable and blocks otherwise.

This execution model performs one-to-one static assignment between ITC nodes and threads, and synchronizes multiple threads by bounded-buffer fireability. It is called "self-timed" because each thread determines the time to fire its own ITC node by itself. By Theorem 8.13, Ω-scheduling for G_{itc} is equivalent to Ω-scheduling for the primitive graph $G^*_{itc} = (V_{itc}, E^*_{itc})$ of G_{itc}, so we can even transform G_{itc} to G^*_{itc} for efficient runtime synchronization — that is, when verifying bounded-buffer fireability, parallel edge sets are now abstracted to primitive edges.

Figure 8.17 presents the *SELF-TIMED-EXECUTION* function that is exe-

cuted by each thread in the self-timed execution model. For a calling thread, the input graph G is either the ITC graph or the primitive version of the ITC graph, and the input node v is the ITC node assigned to the calling thread. In Figure 8.17, we use several multithreading-specific operations that are widely available in multithread APIs, e.g., NSPR (Netscape Portable Runtime) [57] and Pthreads (Portable Operating System Interface threads) [15]. These operations are underlined for emphasis: lock [4] and unlock are used for mutually exclusive access of an object; wait [5] blocks a thread until the condition variable for which it is waiting is signaled; and signal (*signal* in Pthreads and *notify* in NSPR) wakes up all threads that are waiting for the associated condition variable. For more details, we refer the reader to [57, 15].

In the *while* loop in Figure 8.17, we first check whether the given ITC node is bounded-buffer fireable. If the result is true, we fire v, otherwise, we force the thread to wait for the signal indicating state transitions in any surrounding edges of v in line 17. After firing v in line 6, we update the number of tokens $tok(e)$ on each input and output edge e of v. Then for each adjacent node u of v, we signal the thread associated with u — if the thread is waiting, it is woken up to check whether u is bounded-buffer fireable due to v's firing. For synchronization purposes and for

[4] Any thread that attempts to acquire a lock that is held by another thread blocks until the holder of the lock exits.

[5] wait should be called by a thread while the *lock* (*mutex* in Pthreads) associated with the condition variable is locked, and the thread will automatically release the lock while it waits. After a signal is received and a thread is awakened, the lock will be automatically locked for use by the thread.

```
SELF-TIMED-EXECUTION(G,v)
input: a consistent SDF graph G = (V, E), an assigned actor v
 1  while simulation is not terminated
 2    lock v's lock
 3    if v is bounded-buffer-fireable
 4      unlock v's lock
 5      n = min(min_{e∈in(v)} ⌊tok(e)/cns(e)⌋, min_{e∈out(v)} ⌊(buf(e) − tok(e))/prd(e)⌋)
 6      fire v for n times
 7      for each edge e ∈ in(v)
 8        lock e's lock, tok(e) = tok(e) − n × cns(e), unlock e's lock
 9      end
10      for each edge e ∈ out(v)
11        lock e's lock, tok(e) = tok(e) + n × prd(e), unlock e's lock
12      end
13      for each node u ∈ adj(v)
14        lock u's lock, signal u's condition-variable, unlock u's lock
15      end
16    else
17      wait for v's condition-variable to be signaled
18      unlock v's lock
19    end
20  end
```

Figure 8.17: Self-timed multithreaded execution function.

correctness in multithreaded implementation, the lock (*lock* in NSPR and *mutex* in Pthreads) associated with an ITC node is locked when verifying bounded-buffer fireability as well as calling signal and wait operations. In addition, a lock mechanism is also required when updating the state $tok(e)$ on an edge e. To prevent from firing consecutive invocations of v one at a time (which is wasteful of synchronization operations), the number of times n to be repeated atomically for v is determined at runtime in line 5.

8.5.2 Self-Scheduled Multithreaded Execution Model

In multithreaded environments, multithread APIs and operating systems schedule the activities of threads and the usage of processing units. In the self-timed

multithreaded execution model, the number of threads to be scheduled is equal to the number of nodes in an ITC graph, even though the processing units are very limited, e.g., 2 or 4 processing units in current multi-core processors. When the number of fireable ITC nodes is larger than the number of processing units, multithreading APIs and operating systems take responsibility for scheduling. Motivated by this observation, we also develop the *self-scheduled multithreaded execution model* to provide an alternative method for executing ITC graphs in multithreaded environments.

Definition 8.32. Given a consistent ITC graph $G_{itc} = (V_{itc}, E_{itc})$, the *self-scheduled multithreaded execution model* allocates a number of threads equal to the number of processing units. Each thread dynamically selects and executes an ITC node $v \in V_{itc}$ that is bounded-buffer fireable and free for execution (i.e., v is not executed by other thread), and blocks when none of the ITC nodes are bounded-buffer fireable and free for execution.

This execution model performs dynamic assignment between ITC nodes and threads and synchronizes threads based on bounded-buffer fireability. It is called "self-scheduled" because threads perform dynamic assignment by themselves.

Figure 8.18 presents the *SELF-SCHEDULED-EXECUTION* function that is executed by each thread in the self-scheduled execution model. Again, the input graph G is either the ITC graph G_{itc} or its primitive version. Initially before calling this function, for each ITC node $v \in V_{itc}$, if v is bounded-buffer fireable, we push v onto a fireable list L and set v's state to *fireable*, otherwise, we set v's state to

not-fireable. The input list L then contains the ITC nodes that are initially bounded-buffer fireable. Here, the state of an ITC node v is used to verify whether v is in L or whether v is under execution in constant time such that other concurrent threads do not mistakenly re-insert v into L (line 22).

Once execution control enters the *while* loop in line 3, we check whether there are ITC nodes in L. If the result is true, we pop the first ITC node v from L, and execute v for a number of times n that is determined at runtime. If L is empty — i.e., no ITC nodes are bounded-buffer fireable and free for execution — we force the thread to wait for a signal indicating changes in L (line 31). Returning back to line 8, after firing v, we update the number of tokens on input and output edges of v, and examine whether v and whether the adjacent nodes of v are bounded-buffer fireable — this is because state transitions in surrounding edges of v only affect bounded-buffer fireability of v and its adjacent nodes. If they become bounded-buffer fireable, we push them onto L. Finally, we signal the possible changes in L, and if there are threads waiting for fireable ITC nodes, this will wake them up. Again, for synchronization purposes and for correctness in multithreaded implementation, the lock mechanism is applied whenever there is a change of state related to ITC nodes, ITC edges, and the fireable list L.

8.6 Simulation Results

In practical implementation of MSS, estimates of actor execution times are required in order to compute the SRTP value of each actor and the SRTP thresh-

```
SELF-SCHEDULED-EXECUTION(G,L)
input: a consistent SDF graph G = (V, E), a fireable list L
 1  while simulation is not terminated
 2    lock L's lock
 3    if L is not empty
 4      pop the first actor v from L
 5      unlock L's lock
 6      n = min(min_{e∈in(v)} ⌊tok(e)/cns(e)⌋, min_{e∈out(v)} ⌊(buf(e) − tok(e))/prd(e)⌋)
 7      fire v for n times
 8      for each edge e ∈ in(v)
 9        lock e's lock, tok(e) = tok(e) − n × cns(e), unlock e's lock
10      end
11      for each edge e ∈ out(v)
12        lock e's lock, tok(e) = tok(e) + n × prd(e), unlock e's lock
13      end
14      lock v's lock
15      if v is bounded-buffer fireable
16        unlock v's lock, lock L's lock, push v in L, unlock L's lock
17      else
18        set v's state to not-fireable, unlock v's lock
19      end
20      for each node u ∈ adj(v)
21        lock u's lock
22        if u is bounded-buffer fireable and u's state is not-fireable
23          set u's state to fireable, unlock u's lock
24          lock L's lock, push u in L, unlock L's lock
25        else
26          unlock u's lock
27        end
28      end
29      lock L's lock, signal L's condition-variable, unlock L's lock
30    else
31      wait for L's condition-variable to be signaled
32      unlock L's lock
33    end
34  end
```

Figure 8.18: Self-scheduled multithreaded execution function.

old for graph clustering at compile-time. In general, a single actor's functionality may range from simple operations, such as addition, multiplication, etc., to complex operations such as FFT, FIR, etc.. Due to this reason (and also based on our experiments), setting unity actor execution time usually causes unacceptable results. Furthermore, using an actor's production and consumption rates as execution time cost functions also results in poor performance. In our approach, we perform *actor execution time profiling* to collect estimates of actor execution times before scheduling. The profiling process repeatedly runs an actor for a short time and takes an average.

We have implemented and integrated the multithreaded simulation scheduler (MSS) in the Advanced Design System (ADS) from Agilent Technologies [67]. However, the design of MSS is not specific to ADS, and the techniques presented in this thesis can be generally implemented in any simulation tool that incorporates SDF semantics and works in multithreaded environments. Indeed, the definitions, theoretical results, and algorithms have been carefully presented in this thesis in a manner that is not specific to ADS.

Our experimental platform is an Intel dual-core hyper-threading (4 processing units) 3.46 GHz processor with 1GB memory running the Windows XP operating system. We use the NSPR API [57] as the multithread library. In the experiments, we use the following three schedulers: 1) our multithreaded simulation scheduler (MSS), 2) the thread cluster scheduler (TCS) [43] in ADS, and 3) our simulation-oriented scheduler (SOS) [38] (see Chapter 7). As discussed in Chapter 3, TCS was developed previously in ADS for simulation runtime speed-up using multithreaded

execution, and it is the only prior work that we are aware of for multithreaded SDF simulation. Also, as presented in Chapter 7, SOS was developed for joint minimization of time and memory requirements when simulating large-scale and highly multirate SDF graphs in single-processor environments (single-thread execution semantics). We use SOS as the single-thread benchmark scheduler for comparing TCS and MSS to state-of-the-art, single-thread SDF execution methods.

In our experiment with MSS, the parameter M for the SRTP threshold (Equation (8.14)) is set to 32, and the buffer upper bound is set to 4,500,000 tokens. For runtime scheduling in MSS, we use the self-scheduled multithreaded execution model due to its efficiency (see Section 8.5.2).

In the experiments, we include 12 wireless communication designs from Agilent Technologies based on the following standards: WCDMA3G (3GPP), CDMA 2000, WLAN (802.11a and 802.11g), WiMax (WMAN, 802.16e), Digital TV, and EDGE. We collect both execution time and total simulation time results: here, execution time refers to the time spent in executing the graph, and this is the component that can be speed-up by multithreaded execution; *total simulation time* refers to the time spent in overall simulation, including actor profiling, scheduling, buffer allocation, and execution. Table 8.1 presents the average execution time and the average total simulation time of the 12 designs under SOS, TCS, and MSS for three runs. Table 8.1 also presents execution time and total simulation time speed-up for TCS over SOS (SOS/TCS) and for MSS over SOS (SOS/MSS). We plot the speed-up in Figure 8.19 for easy comparison.

As shown in Figure 8.19, MSS outperforms TCS in all designs. MSS can

achieve around 3.5 times execution time speed-up on designs 4, 5, 9, 12, and around 2 to 3 times execution time speed-up on designs 2, 3, 6, 7, 8, 11. Note that the speed-up from MSS is provided by not only the multi-core capability but also the novel clustering and actor vectorization techniques. TCS performs worse than single-thread SOS in designs 1, 6, 7, and 10 due to its un-balanced partitioning, which takes numbers of firings into account rather than SRTP values. Furthermore, TCS encounters out-of-memory problems in design 12 due to its heavy dependence on the cluster loop scheduler, which cannot reliably handle highly multirate SDF graphs (see Chapter 7).

Regarding the total simulation time values in Table 8.1, MSS spends around 2 to 10 seconds more compared to execution time due to overheads in environment setup, actor profiling, scheduling, buffer allocation, and multithreading initialization and termination. In contrast, SOS only requires around 1 to 3 seconds more. Based on our experiments, scheduling time for MSS is similar or even faster than SOS. The overheads from MSS are mostly due to actor profiling, multithreading initialization/termination, and longer buffer allocation (because MSS trades off buffer requirements for synchronization reduction). However, the additional overhead from MSS is insignificant compared to the large simulation times that are observed. For long-term simulations, our results have shown that MSS is a very effective approach to speeding up overall simulation for SDF-based designs.

The speed-ups from MSS for design 1 and 10 are not as significant as for other designs. Based on our investigation into this, the limitations of MSS here may in general come from a number of relevant properties in these designs. First, these de-

signs may contain actors whose SRTP values are larger than the SRTP threshold. In other words, a heavily-computational actor with large repetition count may become a bottleneck in multithreaded execution. Second, these designs may involve actors that require slow or non-parallelizable external resources. For example, slow hard drive reading or writing operations may become bottlenecks. In addition, file readers and writers from parallel threads may compete for hard drive channels. Third, these designs may involve strongly connected components (SCCs) whose SRTP values are larger than the SRTP threshold. This limitation results from SCC clustering in MSS. However, decompositions of SCC clusters may not help in some cases. For example, in a large homogeneous cycle (i.e., a cycle in which production and consumption rates are identically equal to 1) with only a single initial delay, actors can only be executed sequentially. In these cases, clustering SCCs and computing static schedules is in general a more efficient approach.

Investigating techniques to address these limitations and further extend the power of MSS is a useful direction for further work.

8.7 Conclusion

Motivated by the increasing popularity of multi-core processors that provide on-chip, thread-level parallelism, we have proposed multithreaded simulation of synchronous dataflow (SDF) graphs to achieve simulation runtime speed-up. We have illustrated the challenges in scheduling large-scale, highly multirate SDF graphs for multithreaded execution. We have introduced Ω-scheduling and associated through-

Table 8.1: Simulation results.

Design #	Description	Execution / Simulation	Time (seconds)			Speed-up (X)	
			SOS	TCS	MSS	$\frac{SOS}{TCS}$	$\frac{SOS}{MSS}$
1	3GPPFDD_UE	execution	216.34	228.26	148.42	0.95	1.46
	_Rx_Performance	simulation	217.55	230.70	152.46	0.94	1.43
2	WCDMA3G_BS	execution	554.19	387.92	184.90	1.43	3.00
	_Rx_Intermod	simulation	556.11	388.97	195.26	1.43	2.85
3	WCDMA3G_BS	execution	419.04	261.70	139.34	1.60	3.01
	_Rx_Blocking	simulation	420.09	262.64	148.18	1.60	2.83
4	WCDMA3G_UE_Rx	execution	267.60	145.98	74.05	1.83	3.61
	_In_Bank_Blocking	simulation	269.54	146.91	81.41	1.83	3.31
5	CDMA2K_Fwd	execution	760.80	758.82	215.39	1.00	3.53
	RC3AWGN	simulation	761.45	759.57	219.16	1.00	3.47
6	CDMA2K_Rev	execution	639.94	688.16	266.78	0.93	2.40
	RC3AWGN	simulation	640.64	688.84	270.12	0.93	2.37
7	WLAN_80211a_24Mbps	execution	201.97	308.35	72.16	0.66	2.80
	_PN_System	simulation	202.50	308.86	74.25	0.66	2.73
8	WLAN_80211g_CCK	execution	94.61	54.44	41.08	1.74	2.30
	_11Mbps_AWGN_System	simulation	95.14	54.97	42.87	1.73	2.22
9	WMAN_OFDMA	execution	354.22	264.05	97.22	1.34	3.64
	_DL_TxWaveform	simulation	356.95	266.68	103.33	1.34	3.45
10	WMAN_OFDMA	execution	198.75	275.13	102.43	0.72	1.94
	_UL_AWGN_BER	simulation	201.13	277.42	108.58	0.73	1.85
11	Digital TV	execution	190.79	183.46	75.64	1.04	2.52
		simulation	193.36	188.55	80.07	1.03	2.41
12	Edge Signal Source	execution	323.98	N/A	90.64	N/A	3.57
		simulation	324.68	N/A	92.52	N/A	3.51

Figure 8.19: Speed-up: execution time and total simulation time.

put analysis as theoretical foundations in our developments. We have then presented the novel multithreaded simulation scheduler (MSS). The compile-time scheduling approach in MSS strategically integrates graph clustering, actor vectorization, intra-cluster scheduling, and inter-cluster buffering techniques to construct inter-thread communication (ITC) SDF graphs for multithreaded execution. Then the runtime scheduling approach in MSS provides self-timed and self-scheduled multithreaded execution models for efficient execution of ITC graphs in multithreaded environments. Finally, on multithreaded platform equipped with 4 processing units, we have demonstrated up to 3.5 times speed-up in simulating modern wireless communication systems with MSS.

Chapter 9

Conclusion, Current Status, and Future Work

9.1 Conclusion

In this thesis, we have presented the dataflow interchange format (DIF) for integrating dataflow models, techniques, EDA tools, DSP libraries, and embedded processing platforms for DSP system design. In Chapter 4, we propose the dataflow interchange format as a standard language for specifying DSP-oriented dataflow graphs. We have also developed and are continuing to augment the DIF package for experimenting with dataflow models and techniques, and working with DSP applications across the growing family of relevant design tools, libraries, and embedded processing platforms.

In Chapter 5, we have proposed the DIF-based porting methodology as a systematic approach for porting DSP designs across design tools and libraries. With this porting methodology and the porting infrastructure provided in the DIF package, migrating or developing DSP designs across tools and libraries can be achieved efficiently, and this achievement is equivalent to porting DSP designs across the underlying embedded processing platforms that are supported by the tools and libraries.

In Chapter 6, we have presented the DIF-to-C software synthesis framework for automatically generating C-code implementations from high-level dataflow spec-

ifications of DSP systems. Our DIF-to-C framework integrates a significant amount of scheduling, buffering, and code generation techniques, and allows designers to associate dataflow actors with their desired C functions. In other words, our DIF-to-C framework offers a useful link between coarse grain dataflow optimizations and hand-optimized libraries, and provides an efficient way to explore the complex range of trade-offs in DSP software implementation.

In the dataflow simulation context, we have presented the simulation-oriented scheduler (SOS) in Chapter 7 to solve major problems encountered in simulating highly-multirate systems. Our SOS scheduler emphasizes effective, joint minimization of time and memory requirements for simulating critical SDF graphs. We have implemented SOS in Agilent ADS and demonstrated large improvements in terms of scheduling time and memory requirements in simulating real-world, large-scale, and highly-multirate wireless communication systems.

To exploit the trend towards multi-core processors in desktop simulation platforms, we have also presented a multithreaded simulation scheduler (MSS) in Chapter 8 to pursue simulation runtime speed-up through multithreaded execution of SDF graphs on multi-core processors. On an Intel dual-core hyper-threading (4 processing units) processor, our results from MSS implementation in ADS demonstrate up to 3.5 times speed-up in simulating modern wireless communication systems (e.g., WCDMA3G, CDMA 2000, WiMax, EDGE, and DTV).

9.2 Future Work

9.2.1 Dataflow Interchange Format Framework

Building hardware synthesis capability is an interesting future direction. We envision this capability can extend the coverage of the DIF framework to various hardware module libraries and hardware platforms, e.g. FPGAs. We are also investigating the incorporation of new dataflow models and techniques, and are working with several industry and research partners to provide more features in DIF.

9.2.2 Intermediate Actor Library

One limitation of our porting approach arises however when working with a large number of tools: when many tools are involved in the porting space, we need to specify the mapping information for each pair of tools. This requires effort and additional code that grows quadratically with the number of tools that are involved.

The *vector, signal, and image processing library* (VSIPL) [39] is an open source, C-based API that provides various commonly used functions in many areas of signal processing. Motivated by the increasing popularity of VSIPL, we propose an enhanced DIF-based porting approach [32] where VSIPL is abstracted and integrated as an intermediate actor library, and the actor mapping mechanism operates by mapping "to" and "from" the abstract version of VSIPL. In particular, the abstract VSIPL specifies only functional interfaces (i.e., computations and their associated arguments) without limiting any implementation issue. With this new configuration, as illustrated in the right part of Figure 9.1, we reduce the requirement of AIF

Figure 9.1: Original porting approach and the integration of abstract VSIPL.

specifications from $N(N-1)/2$ to N.

9.2.3 Bounded SDF Scheduling

The conventional model for executing SDF graphs assumes that an application graph will execute infinitely (i.e., it will execute iteratively on one or more input data streams that are on indefinite or unbounded length). This assumption is suitable for hardware and software synthesis of DSP applications, and the scheduling problem under this assumption can be formulated as a problem of computing a cost-efficient periodic schedule for repeated execution. In the simulation context, minimal periodic SDF schedules (see Section 2.1.1) are also favorable for long-term simulations or for simulations with specified numbers of iterations. In fact, the SOS approach employs this concept.

However, simulation tools are adaptable to various termination scenarios. For example, Agilent ADS [67] can terminate a simulation when certain user-specified actors have produced or consumed certain numbers of data tokens. This feature is especially beneficial when simulating critical SDF graphs because minimal periodic schedules are often too long compared to the simulation requirements in initial development stage (in contrast to the longer-range simulations that are employed

in the later stages of development).

Motivated by this new concept of simulation termination, *bounded SDF scheduling* can be explored in the future to explicitly take information about simulation termination into account during scheduling and during execution for more efficient SDF simulation.

9.3 Current Status

DIF is being developed in the University of Maryland DSP-CAD Research Group. Currently, DIF is being evaluated and used by a number of research partners, including MCCI, which has developed DIF exporting and importing capabilities in its Autocoding Toolset.

The Advanced Design System (ADS) from Agilent Technologies is a commercial EDA tool used by many research groups and companies. Our simulation-oriented scheduler has been integrated into ADS and provided as an optional scheduler in the ADS 2006 release. Our multithreaded simulation scheduler is being planned for incorporation into the next ADS release.

Bibliography

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets," in *Proceedings of the Design Automation Conference*, June 1997.

[2] H. Andrade and S. Kovner, "Software synthesis from dataflow models for embedded software design in the G programming language and the LabVIEW development environment," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[3] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, Oct. 2001.

[4] S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms," *IEEE Transactions on Signal Processing*, vol. 42, no. 5, pp. 1190–1201, May 1994.

[5] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for DSP," *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, Sept. 2000.

[6] S. S. Bhattacharyya and P. K. Murthy, "The CBP parameter — a module characterization approach for DSP software optimization," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 38, no. 2, pp. 131–146, Sept. 2004.

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[8] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Transactions on Signal Processing*, vol. 45, no. 6, pp. 1605–1618, June 1997.

[9] ——, "Resynchronization for multiprocessor DSP systems," *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, vol. 47, no. 11, pp. 1597–1609, Nov. 2000.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[11] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Dept. of EECS, U. C. Berkeley, Ph.D. Thesis UCB/ERL 93/69, 1993.

[12] ——, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems," in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 1994.

[13] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, vol. 4, pp. 155–182, Apr. 1994.

[14] J. T. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco," in *Proceedings of the International Workshop on Hardware/Software Codesign*, San Diego, CA, May 2000.

[15] D. R. Butenhof, *Programming with POSIX Threads*, 1st ed. Addison-Wesley, 1997.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[17] I. Corretjer, C. Hsu, and S. S. Bhattacharyya, "Configuration and representation of large-scale dataflow graphs using the dataflow interchange format," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Banff, Canada, Oct. 2006, pp. 10–15.

[18] M. Cubric and P. Panangaden, "Minimal memory schedules for dataflow networks," in *Proceedings of CONCUR '93*, Aug. 1993, pp. 368–383.

[19] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen, "Affine nested loop programs and their binary cyclo-static dataflow counterparts," in *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, Steamboat Springs, Colorado, September 2006, pp. 186–190.

[20] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, Sept. 1997.

[21] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[22] E. Gagnon, "SableCC: An object-oriented compiler framework," School of Computer Science, McGill University, Montreal, Canada," Master's Thesis, 1998.

[23] E. R. Gansner, E. Koutsofios, S. C. North, and K. Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, Mar. 1993.

[24] G. R. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved dataflow programs for dsp computation," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, San Francisco, CA, Mar. 1992, pp. 561–564.

[25] M. Geilen and T. Basten, "Reactive process networks," in *Proceedings of the International Workshop on Embedded Software*, September 2004, pp. 137–146.

[26] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the Design Automation Conference*, Anaheim, CA, June 2005, pp. 819–824.

[27] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. D. Man, "DSP specification using the Silage language," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, Albuquerque, NM, Apr. 1990, pp. 1057–1060.

[28] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, 1st ed. Springer, 2002.

[29] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *IEEE Computer*, vol. 30, no. 9, pp. 79–85, Sept. 1997.

[30] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubhr, A. Deyhle, A. Hadert, and J. Teich, "A systemc-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 47 580, 22 pages, 2007.

[31] C. Hsu and S. S. Bhattacharyya, "Dataflow Interchange Format version 0.2," Institute for Advanced Computer Studies, University of Maryland, College Park, Technical Report UMIACS-TR-2004-66, Nov. 2004.

[32] ——, "Integrating VSIPL support in the Dataflow Interchange Format," in *Proceedings of the Annual Workshop on High Performance Embedded Computing*, Lexington, MA, Sept. 2005, pp. 75–76.

[33] ——, "Porting DSP applications across design tools using the Dataflow Interchange Format," in *Proceedings of the International Workshop on Rapid System Prototyping*, Montreal, Canada, June 2005, pp. 40–46.

[34] ——, "Cycle-breaking techniques for scheduling synchronous dataflow graphs," Institute for Advanced Computer Studies, University of Maryland at College Park, Technical Report UMIACS-TR-2007-12, Feb. 2007, also Computer Science Technical Report CS-TR-4859.

[35] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools," in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004, pp. 423–432.

[36] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the Dataflow Interchange Format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, TX, Sept. 2005, pp. 37–49.

[37] C. Hsu, S. Ramasubbu, M. Ko, J. L. Pino, and S. S. Bhattacharyya, "Efficient simulation of critical synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, San Francisco, CA, July 2006, pp. 893–898.

[38] ——, "Efficient simulation of critical synchronous dataflow graphs," *ACM Transactions on Design Automation of Electronic Systems - Special Issue on Demonstrable Software Systems and Hardware Platforms*, 2007, to be published.

[39] R. Janka, R. Judd, J. Lebak, M. Richards, and D. Campbell, "VSIPL: An object-based open standard API for vector, signal, and image processing," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, Salt Lake City, UT, May 2001, pp. 949–952.

[40] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 2006.

[41] V. Kianzad and S. S. Bhattacharyya, "Efficient techniques for clustering and scheduling onto embedded multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, July 2006.

[42] S. J. Kim and J. C. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," in *Proceedings of the International Conference on Parallel Processing*, vol. 3, University Park, PA, 1988.

[43] J. S. Kin and J. L. Pino, "Multithreaded synchronous data flow simulation," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, Mar. 2003.

[44] D. Ko and S. S. Bhattacharyya, "Modeling of block-based DSP systems," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 40, no. 3, pp. 289–299, July 2005.

[45] M. Ko, P. K. Murthy, and S. S. Bhattacharyya, "Compact procedural implementation in DSP software synthesis through recursive graph decomposition," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Amsterdam, The Netherlands, Sept. 2004, pp. 47–61.

[46] M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained block processing optimization for synthesis of DSP software," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 2006, pp. 137–143.

[47] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak, "Optimizing computations for effective block-processing," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 3, pp. 604–630, July 2000.

[48] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: A system-level prototyping environment for DSP applications," *IEEE Computer Magazine*, vol. 28, no. 2, pp. 35–43, Feb. 1995.

[49] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP," in *Proceedings of the IEEE Global Telecommunications Conference*, vol. 2, Dallas, TX, Nov. 1989, pp. 1279–1283.

[50] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya, "Gabriel: A design environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, pp. 1751–1762, Nov. 1989.

[51] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.

[52] E. A. Lee and S. Neuendorffer, "MoML - a modeling markup language in XML, version 0.4," U. C. Berkeley, Technical Report UCB/ERL M00/12, Mar. 2000.

[53] ——, "Actor-oriented models for codesign," in *Formal Methods and Models for System Design: A System Level Perspective*, R. Gupta, P. L. Guernic, S. K. Shukla, and J.-P. Talpin, Eds. Kluwer Academic Publishers, 2004, pp. 33–56.

[54] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[55] *Using The MCCI Autocoding Toolset Overview*, Management Communications and Control, Inc., document Version 0.98a.

[56] P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[57] *NSPR Reference*, Mozilla.org, available: http://www.mozilla.org/projects/nspr/reference/html/index.html.

[58] P. K. Murthy, "Scheduling techniques for synchronous and multidimensional synchronous dataflow," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec. 1996.

[59] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 177–198, Feb. 2001.

[60] ——, *Memory Management for Synthesis of DSP Software*. CRC Press, 2006.

[61] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, "Joint minimization of code and data for synchronous dataflow programs," *Journal of Formal Methods in System Design*, vol. 11, no. 1, pp. 41–70, July 1997.

[62] P. Murthy and E. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, Aug. 2002.

[63] H. Oh, N. Dutt, and S. Ha, "Shift buffering technique for automatic code synthesis from synchronous dataflow graphs," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, New York Metropolitan area, USA, Sept. 2005.

[64] ——, "Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama, Japan, Jan. 2006, pp. 497–502.

[65] H. D. Patel and S. K. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling*. Kluwer Academic Publishers, 2004.

[66] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Nov. 1995, pp. 122–126.

[67] J. L. Pino and K. Kalbasi, "Cosimulating synchronous DSP applications with analog RF circuits," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Nov. 1998.

[68] R. Reiter, "Scheduling parallel computations," *Journal of the Association for Computing Machinery*, Oct. 1968.

[69] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proceedings of the International Conference on Application-Specific Array Processors*, Venice, Italy, Oct. 1993, pp. 285–296.

[70] C. B. Robbins, "Autocoding toolset software tools for automatic generation of parallel application software," Management Communications and Control, Inc.," Technical Report, 2002.

[71] S. Saha, C. Shen, C. Hsu, A. Veeraraghavan, A. Sussman, and S. S. Bhattacharyya, "Model-based OpenMP implementation of a 3D facial pose tracking system," in *Proceedings of the Workshop on Parallel and Distributed Multimedia*, Columbus, Ohio, August 2006.

[72] A. Sangiovanni-Vincentelli, "The tides of EDA," *IEEE Design & Test of Computers*, vol. 20, no. 6, 2003.

[73] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, 1989.

[74] M. Sen, I. Corretjer, F. Haim, S. Saha, J. Schlessman, T. Lv, S. S. Bhattacharyya, and W. Wolf, "Dataflow-based mapping of computer vision algorithms onto FPGAs," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 49 236, 12 pages, 2007, doi:10.1155/2007/49236.

[75] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc., 2000.

[76] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the Compaan/Laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference*, Feb. 2004.

[77] W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration of hardware software codesign workflow in PeaCE," in *Proceedings of International Conference on VLSI and CAD*, Oct. 1997.

[78] J. Teich and S. S. Bhattacharyya, "Analysis of dataflow programs with interval-limited data-rates," in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004, pp. 507–518.

[79] *TMS320C64x Image/Video Processing Library Programmers Reference*, Texas Instruments, Oct. 2003.

[80] *TMS320C67x DSP Library Programmers Reference Guide*, Texas Instruments, Feb. 2003.

[81] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[82] *Extensible Markup Language (XML) 1.0*, 4th ed., World Wide Web Consortium, 2006, available: http://www.w3.org/TR/2006/REC-xml-20060816/.

[83] *CORE Generator Guide, Version 4.1i*, Xilinx, Inc.

[84] G. Zhou, M. Leung, and E. A. Lee, "A code generation framework for actor-oriented models with partial evaluation," Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2007-29, February 2007.

[85] E. Zitzler, J. Teich, and S. S. Bhattacharyya, "Multidimensional exploration of software implementations for DSP algorithms," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 24, no. 1, pp. 83–98, Feb. 2000.